Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVI Ciclo – 2003– V

Publish/Subscribe Communication Systems: from Models to Applications

Antonino Virgillito

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVI Ciclo - 2003– V

Antonino Virgillito

Publish/Subscribe Communication Systems: from Models to Applications

Thesis Committee

Prof. Roberto Baldoni (Advisor)
Prof. Tiziana Catarci
Prof. Daniele Nardi

Reviewers

Dr. Roy Friedman
Prof. Priya Narasimhan

Author's address:
Antonino Virgillito
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: virgi@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼virgi/

Contents

Chapter 1

Introduction

The world-wide connectivity achieved with the explosion of the Internet is now a well established reality. Possibilities are currently ever growing, thanks to the widespread diffusion of high-bandwidth links and of powerful mobile devices such as wireless laptops, palm computers or new generation mobile phones. The result is that millions of users are now potentially able to communicate, with massive loads of information possibly being exchanged from one side to another of networks spanning a world-wide range.

One of the biggest challenges in next-generation distributed computing is represented by large-scale diffusion of information. Example of applications are stock and news tickers, traffic information, instant messaging and electronic auctions. The general model behind these applications is based on gathering information from a set of data sources, and delivering it to all the users, depending on their interest. Such applications are expected to handle a huge number of concurrent users, with frequent information publication and dynamic changes in users' interest.

The design of this type of distributed applications in such a highly demanding context still hides many issues to cope with and the large spectrum of possibilities offered by the technological advances cannot be by themselves the answer. Powerful tools are still required that can effectively exploits available computational resources, carefully avoiding to overuse them more than what strictly necessary. On the other hand, such tools have to provide to both application developers and users a flexibility allowing them a quick usage and an easy deployment in a broad range of situations.

The classical abstractions on which distributed applications have been built until now cannot keep this pace anymore. For example, the common RPC paradigm that is the basis for the most popular middleware tools, have proved to be inadequate for large-scale interactions requiring a frequent diffusion of information among many participants. The reason is that RPC promotes a tight coupling among participants, in the sense that recipients for a piece of information have to be explicitly targeted by its sender. When envisioning a scenario where the set of recipients for a piece of information can be composed by a large number of entities and can frequently change over time, it is easy to understand that tightly coupled communication paradigms such as RPC experience an intrinsic scalability limit.

More appropriate solutions for many-to-many wide-area diffusion are represented by network-level technologies such as IP multicast. However, these are low-level facilities that still needs high-level interfaces for being easily integrated into applications. Moreover, an actual world-wide deployment seems still a long way to come [34] and the management of dynamically changing multicast groups cannot be handled easily.

For this reason, a great attention has been paid in the last years for research focused on distributed solutions targeted to information diffusion specifically for wide-area environments. Among the the most active areas of research in this sense we cite peer-to-peer overlay network infrastructures [105, 116, 95, 89], epidemic multicast algorithms [43] and and, finally, event-based systems following the publish/subscribe paradigm, which are the subject of study of this thesis.

Though publish/subscribe (*pub/sub*) is not a recent achievement [10, 103], its use in large-scale, wide-area communication has become only in the last years a hot research topic, making pub/sub move from a simple application of multicast to a communication paradigm in its own right. This happened because the anonymous, loosely coupled communication scheme that is proper of the pub/sub paradigm, fits well to the highly dynamic nature of large-scale environments. In the following we quickly introduce the main features of the pub/sub paradigm, and then present some open research issues related to it.

1.1 The Publish/Subscribe Paradigm

Each participant in a pub/sub-based communication system can take on the role of a *publisher* or a *subscriber* of information. Publishers produce information, referred in the literature as *notifications* (or *notifications*), which is consumed by subscribers. The main semantical characterization of pub/sub is in the way notifications flow from senders to receivers: receivers are not directly targeted from publisher, but rather they are indirectly addressed according to the content of notifications. That is, a subscriber expresses its interest by issuing *subscriptions* for specific notifications, independently from the publishers that produces them, and then it is *asynchronously* notified for all notifications, submitted by any publisher, that match their subscription. “Asynchronous” means that a subscriber does not have to be blocked waiting for notifications to arrive, such as in client/server RPC, but it can keep on performing concurrent operations.

In order to avoid each publisher to have to know all the subscription for each possible subscriber, this propagation mechanism is realized by introducing a logical intermediary between publishers and subscribers, that in the literature is usually referred to as *Notification Service*. Both publishers and subscribers communicate only with a single entity, the Notification Service, that *(i)* stores all the subscriptions associated with the respective subscribers, *(ii)* receives all the notifications from publishers, *(iii)* dispatches all the published notification to the correct subscribers. The result is that publishers and subscribers exchange information without directly knowing each other. This *anonymity* is one of the main features of the pub/sub paradigm and simply stems from the level of indirection provided by the Notification Service.

Pub/sub is then an anonymous, many-to-many, asynchronous communication paradigm, where multiple producers may propagate information to multiple consumers. Anonymity is an effective solution to easily get scalability at abstraction level. Participants do not have to know each other and when the size of the system grows, they still have to contact only the Notification Service.

1.1.1 Research Challenges for Publish/Subscribe

It is clear that given such a simple yet powerful abstraction, the scalability problems move to the realization of the Notification Service. In other words, the Notification Service should be able to face a large amount of users and to span large-scale communication networks, always maintaining an acceptable level of performance. The realization of a scalable pub/sub system hides several interesting research challenges that have made pub/sub a meeting point of different research communities, such as databases, software engineering and distributed systems.

The area of interest of this work is distributed systems, then our attention will be put on the realization of efficient and scalable distributed algorithms and architectures for wide-area pub/sub interactions. In particular, our interest is focused on distributed implementations of the Notification Service, made up of a set of independent processes that interact among themselves with the common aim of dispatching notifications to all interested subscribers. The lively research in this field during the last years has stimulated many ideas that are yet to be completely followed. Moreover, the many points of view under which this problem can be attacked led to a general confusion, without a common unifying framework allowing to understand and compare the different contributions.

In our opinion, the first real research challenge in pub/sub is building such a common vision, proposing general models and frameworks that precisely capture and describe the peculiarities of the paradigm. Though previous contributions in this direction [40, 76] already represented a great step in positioning and understanding the pub/sub paradigm, this road can be followed further.

On the algorithmic side, the main trigger of pub/sub research has been the attempt to build systems that offer a high flexibility to their users, for example allowing them to precisely characterize their interest with powerful and expressive subscription languages. Such systems are referred to as *Content-based* pub/sub systems. Content-based pub/sub obviously requires the Notification Service to rely on complex mechanisms and the big challenge is to build them in a scalable way [18]. Several systems and algorithms addressing these issues have been proposed, for example for efficiently matching notifications against a large number of subscriptions [1, 16, 44] or efficiently delivering them to a large number of users [7, 20]. However, the application of such systems is still restricted to the research community and practical experiences are still lacking. In other words, though research results in content-based pub/sub are now consolidated, actual deployments still rely on more simple, but more efficient solutions.

There are two requirements that have to be satisfied: First, pushing the scalability limits of pub/sub one step forward, by devising new solutions; Second, care about those aspects related to the ease of deployment providing system with dynamic self-organization capabilities. The recent research contributions are following these guidelines, in particular by borrowing and exploiting results achieved in the research on peer-to-peer overlay network infrastructures [117, 22, 108, 84]. This mixture can produce a whole bunch of new ideas that can completely give a new face to this research area. Results obtained by following this direction could also represent a strong basis to cope with the new challenges represented by the application of pub/sub system in highly dynamic scenarios, such as those comprising mobile devices.

1.2 Contributions of the Thesis

This thesis presents the results of a broad-range study on research problems related to the application of the publish/subscribe paradigm in wide-area network. The first contribution is a general survey of the state of the art of research in pub/sub area. The presentation ranges from a general description of the paradigm to a deep investigation of the internal architecture of a distributed pub/sub system, presenting all the issues hidden behind the realization of a scalable pub/sub system, together with the possible solutions that have been proposed in the literature.

Differently from other previous surveys [40, 21, 76], our proposal does not intends to characterize the pub/sub paradigm with respect to other distributed abstractions, but rather to give a wide-range "internal" classification of the pub/sub research area, clarifying also the relationships with other research problems. The other contributions of this thesis can be divided in three general areas: models, regarding the formalization of several aspects of pub/sub, algorithms, presenting novel solutions and applications for pub/sub, and applications, describing a context-specific pub/sub design.

Models: Semantics and Performance of a Pub/Sub System. Currently, only one specific research contribution [76] has been devoted to the formal specification of pub/sub system. Nevertheless, the decoupled nature of this paradigm hides many subtleties that still make necessary a further, deeper reasoning about the precise semantics of a pub/sub interaction. We propose a computational model that characterizes the semantics of a pub/sub system in terms of the classical safety and liveness properties of a distributed system. These properties, however, are expressed basing on two time delays, required to model the decoupled nature of the paradigm.

We also show how the non-determinism introduced by the decoupling may provoke information not to be delivered on time to all the intended subscribers. Thus, following the computational model, an analytical model is introduced that characterizes the performance of a pub/sub system, in terms of the fraction of notifications that successfully reach their destinations.

A detailed analytical study has been carried out, that captures the behavior of a wide class of pub/sub systems. A simulation study, realized by implementing a complete pub/sub system prototype, provides a validation for the analytical results.

Algorithms: Self-organization in Content-based Pub/Sub Systems. As pointed out above, content-based pub/sub systems have been the main inspiration for research on scalable algorithms for notification diffusion. Common content-based pub/sub systems are built over distributed application-level networks of notification brokers, acting as servers for both subscribers and publishers. The links among brokers in these networks are in practice static TCP connections, made up at system creation time (generally assuming a human intervention). The idea behind our contribution is to try to push the scalability limit of content-based pub/sub by introducing the possibility of rearranging the application-level network topology. In particular, brokers are able to self-organize their connections according to the distribution of interest among their subscribers. This creates paths composed only by brokers serving subscribers which are interested in the same information, avoiding to involve brokers that exclusively carry out a forwarding function.

A basic algorithm for self-organizing content-based pub/sub is first presented. This algorithm achieves performance results very close to an ideal value: after the self-organization the number of brokers involved in a notification diffusion is almost equal to the number of the ones interested in the notification itself, i.e. the minimum possible. Furthermore, a variant of the basic algorithm is presented that also accounts for the impact of self-organization on the performance metrics in the underlying network.

Applications: Pub/sub for Data Quality Notification. The real-world usage of pub/sub systems have been described in several application contexts. We present a novel application of a pub/sub-based service realized as a part of a research project that involves the research areas of information systems, databases and distributed systems. The problem attacked by this project is the management of data quality [90] in those systems formed by different, independent information systems, cooperating to achieve common goals (namely, Cooperative Information Systems). In particular, we present the design of a pub/sub service, aimed at the notification of changes in quality of data. The design of the service includes novel solutions to tackle the scalability and integration issues arising from the cooperative scenario. The design and the implementation (based on the web-services technology) of the service are presented.

1.3 Structure of the Thesis

The following is an outline of the content of the thesis:

In Chapter 2 we first give a general, high-level specification of a pub/sub communication system, then we survey the state of the art in this research field, by first giving a wide-range classification of all the possible general solutions to the aforementioned problems and then by presenting how such problems are solved in actual systems.

In Chapter 3, a more specific, formal description of the semantics of a pub/sub system is given. A probabilistic analytical model is also presented, validated through a simulation study.

In Chapter 4 we address scalability issues underlying a pub/sub implementation, by proposing self-organization algorithms for distributed content-based pub/sub system.

Chapter 5 presents the design and the implementation of a distributed pub/sub system (namely, the Quality Notification Service), specifically designed for being used in cooperative information systems for managing data quality issues.

Chapter 2

Understanding Publish/Subscribe Systems

The general objective of a publish/subscribe (pub/sub) system is to let information propagate from publishers to interested subscribers, in an anonymous, decoupled fashion. Such a common general behavior is implemented with different flavors in actual systems known in the pub/sub literature. In particular, the three aspects that have to be specified are:

❒ How subscribers' interest is expressed in relation to information. In other words, which is the query language used by subscribers for issuing subscriptions to the Notification Service.

❒ How the Notification Service is implemented. The Notification Service can be realized as a single, centralized entity or as a distributed set of processes.

❒ How information is propagated from publishers to subscribers. That is, how the Notification Service exploits the underlying network levels in order to correctly dispatch information to interested subscribers.

These issues are tightly coupled with one another and their combination strongly influences the mechanisms underlying the pub/sub system. For example, a simple subscription language can favor the implementation of the diffusion mechanism but as a drawback provides a low expressive power for users to express their interest. The pub/sub paradigm has inspired a great amount of research in recent years trying to achieve trade-offs between these conflicting issues. However, this research area currently lacks a unifying view in which each research contribution can be exactly positioned.

In this Chapter we describe all the problems hidden behind a distributed pub/sub system and present a framework to classify the different ways for them

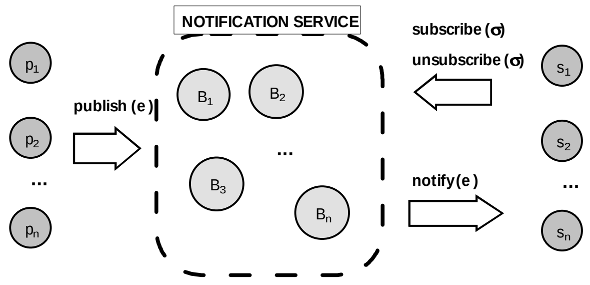


Figure 2.1: High-level view of a pub/sub system.

to be addressed that have been proposed in the pub/sub literature. In the first part of the chapter we do not refer to any specific system or implementation, but consider problems at their most general level. This allows us to isolate the overall issues from what are specific implementations techniques. In the final part of the chapter we survey the most representative pub/sub systems, positioning them with respect to the general reference framework. At the best of our knowledge, this is the first attempt to realize a general survey of the area that captures all the current state-of-the-art systems and research contributions.

2.1 Basic Publish/Subscribe Specification

In this section we propose a general high-level framework of a pub/sub system, by first describing the participants to the system and their roles and then defining the various aspects of their interaction. A formal specification of the semantics of a pub/sub system is given in Chapter 3.

2.1.1 Elements of a Publish/Subscribe System

A generic pub/sub communication system (PSS) can be represented by a triple $< \Pi, B, \Sigma >$ of sets of processes (Figure 2.1). Sets in the triple are defined depending on the role of the processes in the system: $\Pi = p_1, \dots, p_n$ is a set of n processes, called the *publishers*, which are producers of information. $\Sigma = s_1, \dots, s_m$ is a set of m processes called the *subscribers*, which are consumers of information. Σ and Π may have a non-zero intersection, that is a same process may act both as a publisher and as a subscriber. $\Delta = B_1, \dots, B_o$ is a set of o processes, called *brokers*.

We assume publishers and subscribers to be *decoupled*: a process in Π cannot communicate directly with a process in Σ and vice versa (unless it is the same process). Decoupling is a desirable characteristic for a communication system because applications can be made more independent from the communication issues, avoiding to deal with aspects such as addressing or synchronization [40]. We discuss this point in Section 2.1.2. Processes in Π and Σ can exclusively communicate with any other process in Δ. Then, the set of brokers Δ represents a logically centralized entity that allows the communication between publishers and subscribers, at the same time maintaining them decoupled. *Delta* in its whole constitute what in literature is often referred to as *Notification Service* (or *Event Service*). Publishers and subscribers act as *clients* for the Notification Service.

In the particular case of $|\Delta| = 1$, we have a centralized implementation for the Notification Service. Centralized implementations are obviously the simplest implementation solution for a Notification Service. However, scalability is limited by the processing power of the machine that hosts the service and its networking resources. In the general case of $|\Delta| > 1$ the Notification Service is implemented as a network of distributed *brokers*. In a distributed implementation, publishers and subscribers can contact indifferently any of the brokers, that share the load of managing subscriptions and publications. The realization of this solution is more challenging, requiring complex protocols for the coordination of the various brokers and the diffusion of the information (discussed in Section 2.4). From now on, we exclusively focus on distributed implementations.

Client Interaction The interaction between client processes and the Notification Service takes place through a set of operations that can be executed by client processes on the Notification Service and viceversa (Figure 2.1). A publisher submits a piece of information e to other processes by executing the `publish(`e`)` operation on the Notification Service. The Notification Service dispatches a piece of information e submitted by other processes to a subscriber by executing the `notify(`e`)` on it. A subscription σ is respectively installed and removed on the Notification Service by subscriber processes by executing the `subscribe(`σ`)` and `unsubscribe(`σ`)` operations.

Notifications and Subscriptions In the following we specify the nature of the information exchanged between publishers and subscribers. Such information is produced in form of *notifications*. In the pub/sub literature also the terms *event* and *publication* are often used, but is important to point out that the three terms are not interchangeable. We clarify the exact way in which these terms are generally used: a publisher produces a *event* (or a *publication*), while the Notification Service issues the corresponding *notification* on subscribers. For simplicity, in the following we only use the term "notification", except where it is important to point out the difference.

The common data model used in pub/sub systems defines a notification as a set of attribute-value pairs. Each attribute has a *name*, a simple character string, and a *type*. The type is generally one of the common primitive data types defined in programming languages or query languages (e.g. integer, real, string, etc.).

On the subscribers' side, interest in specific notifications is expressed through *subscriptions*. A subscription is a pair $\sigma = (f, s)$, where $s \in \Sigma$ is the subscriber which is interested in notifications declared through the *filter* f. We say a notification e *matches* a subscription σ if it satisfies its filter f. The task of verifying whenever a notification e matches a filter f is called *matching* ($e \sqsubset f$). The precise characterization of the possible format of f is the subject of Section 2.2.

2.1.2 Positioning the Publish/Subscribe Paradigm

The simple model of pub/sub presented above highlights the characterizing aspects of this paradigm, that is the decoupling among participants and the many-to-many interaction. Such features are desirable properties for building scalable distributed applications, but are not offered by any of the other common distributed communication paradigms. In the following we give a brief presentation of the most popular paradigms for realizing distributed interactions, pointing out the differences with the pub/sub paradigm. A detailed comparison among pub/sub and other paradigms can be found in [40].

Remote Procedure Calls. Remote Procedure Calls [14] represent the first basic form of abstraction of a distributed computation, extending the common sub-program invocation to a distributed level, by wrapping transparently the aspects related to remote communication. RPC is probably also the most popular distributed paradigm, thanks to the different form it has been presented. RPC-based mechanisms are in fact included in the C and Java [72] languages, and are the foundation of the most popular middleware technologies, such as CORBA [50], DCOM [85] and J2EE [74]. Finally, the SOAP protocol [27], foundation of the Web Services technology [79], is the latest incarnation of RPC. RPC realizes basically a one-to-one interaction, with a strong coupling among the participants: the caller must own a reference to each entity it wants to communicate with and a many-to-many interaction is difficult to realize in an efficient way. Moreover, the interaction is generally completely synchronous: the called entity act as a server, and must remain available for being invoked for its entire lifetime, while the calling entity act as a client, that generally must remain blocked until it receives a reply from the server (though asynchronous invocations are common, where the client can leave the communication without waiting for the result).

Shared Spaces. Shared spaces has been the first paradigm to consider an indirect communication. This is realized by a distributed shared memory, common to all participants, that interact each other by writing and reading data from/to the shared space. Actually, this realizes a many-to-many anonymous interaction, where many producers can indirectly send messages that will be received by many consumers. The difference with pub/sub is that consumers are not asynchronously notified but retrieve messages in a push-style fashion with an explicit, synchronous request. Among the most popular shared space implementations we cite Linda [47], JavaSpaces [46] and TSpaces [64].

Message Queues. Maybe the most common alternative to pub/sub for realizing interactions with multiple recipients is the message queue paradigm. Message Queues are an abstraction that is particularly used in the industry, with many popular existing implementations, such as IBM WebSphereMQ [58], Microsoft Message Queue [85] and part of the JMS specification [74]. The reason is that the message queue paradigm can easily provide transactional or reliability guarantees, thanks to the fact that messages are persistently stored within the queues. Moreover, it is often used as the basis for asynchronous invocation to software components (such as COM+ Queued Components, Message-Driven Enterprise Java Beans or Web Services). All the communications in this paradigm are filtered by the queue, that covers a role similar to the Notification Service in pub/sub. The difference is that each participant may have its own queue and a one-to-many interaction could require addressing several queues. Another feature that makes the level of decoupling obtained through message queue lower than the one provided by pub/sub is the fact that consumers must explicitly pull the messages from the queue. However, push-style callback is often present, also with a one-to-many delivery, making the message queue paradigm similar to a persistent form of pub/sub.

2.2 Subscription Models

Different ways for specifying the notifications of interest have led to identifying different variants of the pub/sub paradigm. Several subscription models appear in the literature, characterized by different expressive powers. Highly expressive models offer to subscribers the possibility to precisely match their interest, i.e. receiving only the notifications they are interested in. However, as we will point out, the expressive power of the subscription language is not simply related to the flexibility of interaction with clients, but has a strong influence on the realization of the whole Notification Service. In this Section we present the most popular pub/sub subscription models highlighting the trade-offs among expressiveness and ease of realizing scalable implementations.

2.2.1 Topic-based Model

In the *topic-based* model notifications are grouped in topics (or subjects) i.e., a subscriber declares its interest for a particular topic and will receive all notifications related to that topic. In other words, the filter $\sigma.f$ of a subscription σ is simply the specification of a topic. Each topic corresponds to a logical *event channel* ideally connecting each possible publisher to all interested subscribers. That is, there exists a static association between a channel and all its subscribers, then when a notification is published, the system does not have to calculate all the receivers. Topic-based model has been the solution adopted in all early pub/sub incarnations. Examples of systems that fall under this category are TIB/RV [77], iBus [3], SCRIBE [22], Bayeux [117] and the CORBA Notification Service [49].

Topics are equivalent to the notion of groups used for instance in the context of group communication [86] (e.g., for replication). This equivalence is not very surprising, since the first systems to offer a form of publish/subscribe interaction were actually extensions of group communication toolkits [25, 12] and the subscription scheme was thus inherently based on groups [10]. Subsequently, subscribing to a topic can be viewed as becoming member of a group and publishing a notification for a topic translates accordingly to broadcasting that notification among the members of the corresponding group. Thanks to the topic-group equivalence the topic-based solution mechanism can drive to very efficient implementations, exploiting directly on one hand the large amount of research work in the multicast area, and on the other the network level multicast implementations for diffusing notifications.

The main drawback of the topic-based model is the very limited expressiveness it offers to subscribers. Let us consider, for example an application managing stock quotes. Though notifications may be structured to contain several attributes (for example, the name of the quote, its current value, its variation), only one attribute may be chosen as being selective for the delivery of notifications (i.e. the topic). In the example, it may be the quote name. As a consequence, a subscriber interested in a subset of notifications related to a specific quote (for example only those signalling a rise of the quote above a certain value) will receive also all the other notifications that belong to the same topic.

To address problems related to low expressiveness of topics, several solutions are exploited in pub/sub implementations. For example, the topic-based model is often extended to provide hierarchical organization of topics, instead of a simple flat structure of the topic space (such as in [49])[1]. A topic B can

[1]Sometimes, the word subject is used to refer to hierarchical topics instead of being simply

be then defined as a sub-topic of an existing topic A. Notifications matching B will be received by all clients subscribed to both A and B. Implementations also often include convenience operators, such as wildcard characters, for subscribing to more than one topic with a single subscription. Though these techniques give the application developer effective solutions to overcome expressiveness limitations of the topic-based scheme, they does not still alter the very nature of the topic concept, i.e. a simple organization of subscribers into group with the great advantage of a simple and efficient implementation, that however may not be sufficient with those applications where the interest of subscribers presents a high variability and cannot be clustered with simplicity.

2.2.2 Content-based Model

In the *content-based* variant, subscribers express their interest by specifying conditions over the content of notifications they want to receive. In other words, a filter in a subscription is a query composed by a conjunction of constraints over the values of attributes of the notification[2]. Possible constraints depend on the attribute type and on the subscription language. Most subscription languages comprise equality and comparison operators as well as regular expressions [20, 100, 44]. Generally constraints can be joined inside filters through AND/OR expressions[3] A complete specification of content-based subscription models can be found in [75]. Examples of systems that fall under the content-based category are Gryphon [53], SIENA [102], JEDI [29], LeSubscribe [87], Ready [52], Hermes [83], Elvin [99].

As an example of the content-based model, let us consider again notifications representing stock quotes. Differently from the topic-based scheme, a subscription can involve *all* the attributes of the notification, on which a subscriber can express a constraint with type-specific operators:

```
StockName = ‘IBM’ and change < -3
StockName = ‘M*’ and change >= 1
```

In content-based publish/subscribe, notifications are not classified according to some pre-defined external criterion (i.e., topic name), but rather according to properties of the notifications themselves, that assume different values in each different notification. As a consequence the set of receivers for each notification cannot be determined a priori but has to be computed at publication time. Then, the higher expressive power of content-based pub/sub comes

a synonymous for topic.

[2]disjunctive constraints can be treated as separate subscriptions

[3]The complexity of the subscription language obviously influences the complexity of matching operation. Then it is not common to have subscription languages allowing queries more complex than ones in conjunctive forms. One example can be found in [16].

at the price of the higher resource consumption needed to calculate for each published notification the set of interested subscribers [19, 39].

It is straightforward to see that a topic-based scheme can be possibly emulated through a content-based one, simply considering filters comprising a single equality constraint. The opposite is not true: in particular, the channel abstraction in the topic-based scheme cannot represent flexible features of the content-based scheme such as comparison operators or complex conjunctive subscriptions [68]. At most, some systems (such as the COM+ Notification Service [85] and the CORBA Notification Service [51]) provide an additional content-based subscription language that allows to filter out notifications received from a channel. We refer to this model as the *filtered topic* model, making a clear distinction with the content-based model.

Stressing the difference between the filtered topic approach and a pure content-based one allows us to clarify once more the implications of the subscription model over the system implementation. The Notification Service exploits subscriptions to derive the set of clients to which the notification must be sent to. In a pure content-based system, a notification that does not match any subscription is not sent to any client, saving network resources. On the other hand, in a filtered topic-based system, recipients can only be determined according to the topic they subscribed to. Only once a notification is sent to a subscribing client, the content-based filter is used to determine if it should be actually delivered to it. If the notification matches a topic but does not match the content-based filter, it is not delivered to the client, generating useless network traffic. Then the higher expressiveness of content-based model can aid to save network resources, sending a notification to all and only the actual subscribers. However, as we will see in Section 2.4, a content-based system requires more information to be sent on the network for determining the recipient set.

In overall, we can say that content-based pub/sub is the most general subscription model. This is the reason why this model has gained a lot of attention from the research community and currently still represents the main source of open problems in the field, especially related to efficient and scalable matching and diffusion [6].

2.2.3 Type-based Model

The third alternative subscription model proposed in the literature is the *type-based* model [37]. The type-based variant enhances common pub/sub with concepts derived from object-oriented programming: notifications are declared as objects belonging to a specific type (*obvents*), which can thus encapsulate attributes as well as methods. With respect to simple, unstructured models, Types represent a more robust data model for application developer providing

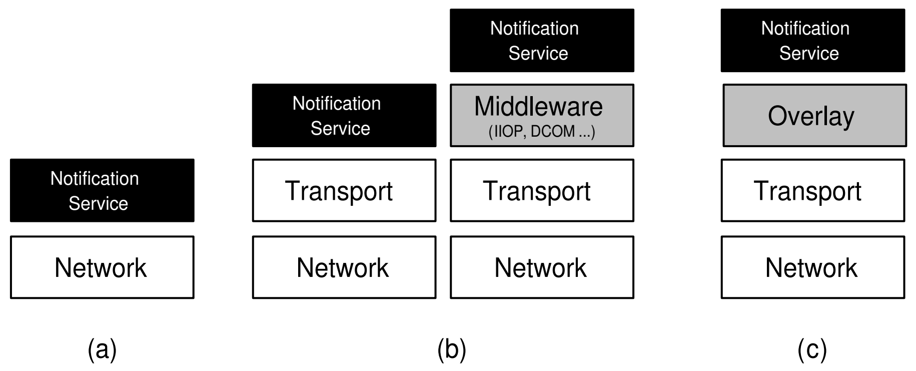


Figure 2.2: Pub/sub Architectural Models

type-safety to be checked by the Notification Service, rather than inside the application [41].

In a type-based subscription the declaration of a desired type is the main discriminating attribute. That is, with respect to the aforementioned models, type-based pub/sub poses itself somehow in the middle, by giving a coarse-grained structure on notifications (like in topic-based) on which fine-grained constraints can be expressed over attributes (like in content-based) or over methods (as a consequence of the object-oriented approach). Type-based pub/sub in this sense resembles the filtered topic model.

Type-based pub/sub was firstly proposed in [42] and fully developed in [37], where it is described an extension to Java for the management of distributed obvents and type-based subscriptions.

2.3 Architectural Models

In this Section we focus on another basic aspect of a pub/sub system, that is the architecture of a distributed Notification Service, with respect to how the various parts it is composed of are structured and what is the mechanism they used to communicate. There are basically three solutions for realizing in practice the architecture of a Notification Service (Figure 2.2):

❒ Relying on multicasting facilities provided by the underlying network levels - Figure 2.2 (a).

❒ Implementing a broker-level notification routing protocol - Figure 2.2 (b).

❒ Exploiting a peer-to-peer overlay network infrastructure for application-level multicasting - Figure 2.2 (c).

In the following we discuss these three solutions, pointing out the drawbacks due to each of them.

2.3.1 Network Multicasting

As pointed out in previous Section, the use of a network-level multicast protocol [32] as a communication layer for the Notification Service has been the natural choice in early systems as they were substantially topic-based. The main advantage of a network-level approach is that it is the easier way to realize many-to-many diffusion experiencing low latencies and high throughput, thanks to the small delays introduced by implementing the protocols exclusively involving routers and switches.

We recall again that multicasting can be directly used in topic-based systems, as each topic corresponds exactly to one multicast group. Using multicasting for content-based systems is not as straightforward because subscribers cannot be directly mapped to multicast groups [78]: This problem has inspired some research work [91, 92, 54], aiming at finding the best possible configuration for multicast groups from a given set of subscribers, where "best" means structuring clusters of subscribers with "similar" subscriptions, so that a notification sent to a multicast group will be delivered to most of its members.

The main drawback of multicast when applied to wide-area scenarios is in its lack of a widespread deployment [34, 101]. Hence, network-level multicasting cannot in general be considered as a feasible solution for applications deployed over a WAN (for example TIB/RV or the CORBA Notification Service uses multicast only for diffusing notifications inside a local area network).

Another undesirable property of network-level multicasting is the lack of reliability guarantees. Several algorithms [111, 56, 45] have been proposed offering reliable delivery through a retransmission mechanism over the unreliable multicast transport. However, this approach has proven not to reach the levels of scalability required in a large-scale pub/sub system, because of the high overhead of message retransmission, making it suitable only for systems with up to a few hundred participants [13].

For this reason, multicast algorithms being designed specifically for being applied to pub/sub systems often belong to the class of gossip algorithms [43]. Gossip (or epidemic or probabilistic) algorithms achieve reliability in a probabilistic sense [13], by guaranteeing that all participants in the system receive any message only with a certain, quantifiable probability. Gossip algorithms reach high level of scalability at the price of a small loss in reliability. Specific gossip algorithms for pub/sub systems have been proposed in [36, 38, 28].

2.3.2 Application-level Networks

The most common approach for the design of a distributed Notification Service is building an application-level network of brokers. Brokers communicate through links consisting of connections over an underlying transport protocol. The underlying broker-to-broker protocol can be then any transport-level or application-level protocol: the most common choice is TCP/IP but also HTTP or middleware protocols such as IIOP or DCOM can be used.

The application-level network is a pure abstraction as links are not required to represent permanent, long-lived connections. The main implication of structuring a Notification Service as an application-level network is that a broker "knows" only a limited set of brokers (i.e., its neighbors in the network) that are the only processes with which it can actually communicate. This allows the system to achieve high degrees of scalability because even if the system size grows, the number of neighbors for each broker remains fixed ensuring that it will manage a bounded number of concurrent open connections and data structures.

The choice of structuring a Notification Service as an application-level broker network is the most common one in actual pub/sub implementations, used by system such as TIB/RV [77], Gryphon [53], SIENA [102], or JEDI [29].

Apart from the application-level routing protocols, that we will analyze in Section 2.4, the main aspect to be clarified in an architecture based on an application-level network of brokers is the topology formed by the brokers themselves. There are basically two solutions, hierarchical or peer-to-peer. In a hierarchical topology, brokers are organized in tree structures, where subscribers' access points lie at the bottom and publishers' access points are roots (or viceversa). Many contributions [9, 110] rely on this topology, thanks to the simplifications it can allow since notifications are diffused only in one direction. However, the hierarchical architecture lacks generality because it relies on a fixed, given structure that has to be specifically made up by application developer and can be hardly modified (e.g. when adding or removing brokers). Moreover in [20], a simulation study shows its inherent inefficiency, due to the fact that brokers belonging to upper levels of the hierarchy experience a higher load than ones at lower levels. In a generic peer-to-peer topology, a broker can be connected with any other broker, with no restrictions. [20] shows the more effective load-balance obtained with respect to a hierarchical topology, though peer-to-peer algorithms may be more difficult to realize. Then, implementations typically use an acyclic topology to aid the routing process.

The main problem of an application-level broker network is the creation of the topology itself. None of the aforementioned system is able to self-organize the broker's network and it is up to the user to decide and set up the connections among brokers. This may generate a non-optimized behavior because the application-level topology may not reflect the underlying physical network: then a single link between two brokers can actually map to a "long" network path, in terms of latency and/or throughput and this may seriously harm the overall performance of the whole system. Furthermore, an application-level communication protocol is inherently slower than one implemented at the router level and, even with the high power of current devices and networks, it is impossible to achieve the same performance levels.

2.3.3 Peer-to-peer Overlay Network Infrastructures

A peer-to-peer overlay network infrastructure realizes an application-level network for information diffusion. It is composed by a set of nodes, each having a unique identifier, and provide the possibility of sending/retrieving information to/from one or more specific node(s), just by specifying their identifier. In other words, they realize a general-purpose unicast or multicast communication facility among the nodes. The main advantage of using such infrastructures in a large-scale setting is their self-organization capability, allowing them to structure the network when a node leaves (for example after a fault) or joins (for example to face a higher number of users). Overlay network infrastructures are then an effective way for easily realizing a large-scale information propagation and have become a widely popular research area in recent years. As a consequence of that, many systems have been developed: we cite among the others Pastry [95], Chord [105], Tapestry [116] (unicast diffusion) or CAN [89], I3 [104] and Astrolabe [109] (multicast diffusion).

Structuring a pub/sub system over an overlay network infrastructure means leveraging the self-organization capabilities of the infrastructure, by building a pub/sub interface over it. The pub/sub behavior is realized through the communication primitives provided by the underlying overlay. With respect to directly building an application-level network of broker, this solution allows to more easily manage dynamic aspects of the systems such as faults and a growing number of broker, inheriting such features from the overlay network infrastructure[4]. Examples of systems using this solution are Bayeux [117] and Scribe [94], for what concerns topic-based systems, and Hermes [84] and Rebeca [108], for what concerns content-based systems. Finally, we cite Select-Cast [15], a multicast system built on top of Astrolabe providing a SQL-like

[4]Rigorously, also an application-level network of brokers does realize an overlay network. Anyway in the literature, the precise nature of an overlay network infrastructure consists in just the basic unicast/multicast behavior, rather than the more specific many-to-many, anonymous semantics of a pub/sub system. This explains the distinction we made between the two solutions.

syntax for expressing subscriptions.

2.4 Behind the Scenes of a Distributed Notification Service

In the previous sections we presented “external” aspects, that is how a pub/sub system can expose its feature to users (Subscription Models) and how it is structured (Architectural Models). In this section we focus on an “internal” view of a pub/sub system, describing what are the mechanisms and algorithms that a pub/sub system must implement, given a subscription model and an architectural model, in order to realize its functionality.

2.4.1 Overview

As pointed out above, we consider distributed Notification Services where each subscriber or publisher can contact any notification broker in order to participate to the system. The brokers to which a subscriber s actually connects are called *access points* for s. The set of access points of a subscriber s is denoted as $AP(s)$. The access point represents only the broker through which s issues subscriptions and receives notifications. In general, the access point may not necessarily host the subscription of its subscribers: for load-balancing purposes or for simplifying notification propagation, a subscription σ may be stored in one or more brokers different from the access point of the issuing subscriber, $\sigma.s$. In this case, these brokers will be referred to as the *target brokers* for σ.

A *subscription configuration* is a set $sc = (\sigma_1, \ldots \sigma_m)$ that contains all the subscriptions present in the whole system at a particular time. The set of all possible subscription configurations is denoted as SC. Finally, we denote the set of all possible notifications as Ω.

The functionality of a distributed Notification Service, in its most most general form, can be thought of as decomposed in the following sub-problems[5] (Figure 2.3):

Event Matching : the task of computing the set of interested subscribers.

Subscription Assignment : identification of a policy used to determine how to distribute subscriptions among brokers.

Subscription Routing : the process of dispatching the subscription from the access point to the target broker, i.e. the implementation of the assignment policy.

[5]In this part of the Chapter we consider the terms events and notifications as not synonyms.

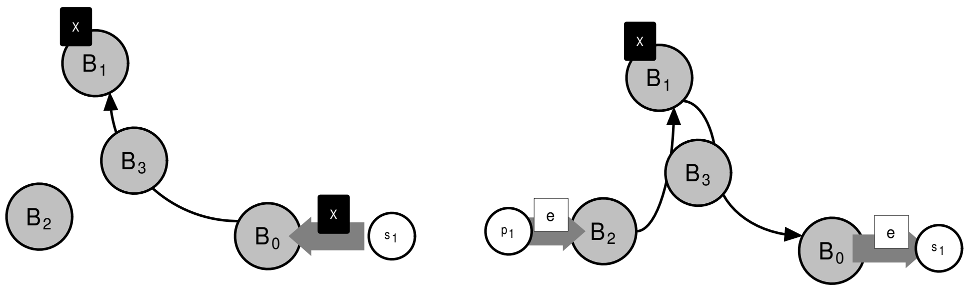


Figure 2.3: Publish/Subscribe Mechanisms

Event Routing : the process of *(i)* identifying the target brokers for the notification (resolving), *(ii)* forwarding the notification through the brokers network in order to reach all possible target brokers

Notification Routing : the process of dispatching a notification to all matching subscribers, i.e. delivering a matching notification from target brokers to the corresponding subscribers.

Figure 2.3 (a) shows an example of subscription assignment and routing: we suppose a partitioning policy that assigns subscription X to broker B_1. When X is issued by a subscriber s_1, it has to be routed from s_1's access point B_0 to target broker B_1. Figure 2.3 (b) shows what happens in the same example situation, when a notification e is published by a client p_1 at broker B_2. The assignment policy applied to e, resolves the notification to target broker B_0. Then, e has to be routed to B_0 where it can be matched (event routing). If e matches X, the corresponding notification has to be routed to interested subscriber s_1 (notification routing). Events and subscriptions are represented in the Figures as two opposite flows, unified by the choice of the target broker given by the partitioning policy. In the following, we provide details on each of the identified mechanisms by discussing their relationships and trade-offs.

We point out that the whole process takes place as a coordination among brokers. Client processes are only the last link of the chain and act in the system only through their access points. Then in the following we do not consider client processes but restrict all the analysis to the network of brokers.

2.4.2 Event Matching

Event matching is an extension of the matching operation defined in Section 2.1.1, that is calculating if a notification satisfies a filter. In this case, the notification has to be matched against *all* the filters in *sc*, returning all the corresponding subscribers. This can be formally represented by defining the following function:

$$\pi : \Omega \times SC \rightarrow 2^{\Sigma}$$

The realization of π is one central and challenging point: as we are dealing with large-scale systems, we expect on one side the overall number of subscriptions in the system to be very high, and on the other a high rate of notifications. Then, the matching operation has to be performed often and on massive data sizes. While obviously this poses no problems in a topic-based system, where matching reduces to a simple table lookup, it is a fundamental issue for the overall performance of a content-based system. The trivial solution of testing sequentially each subscription against the notification to be matched may result in a very poor performance in this settings.

Techniques for efficiently performing the matching operation are then one important research issue related in the pub/sub field. Since they are more related to other research fields rather than distributed computing (e.g. active databases), we only give a brief survey on the solutions actually exploited in content-based implementations.

These can be grouped in two main categories [93], namely *predicate indexing* algorithms and *testing network* algorithms. Predicate indexing algorithms are structured in two phases: the first phase is used to decompose filters of subscriptions into elementary constraints and determine which constraints are satisfied by the notification; in the second phase the results of the first phase are used to determine the filters in which all constraints match the notification. Matching algorithms falling into the predicate indexing family are [80, 44]. Testing network algorithms ([1, 48, 16]) are based on a pre-processing of the set of subscriptions that builds a data structure (a tree in [1] and [48] or a binary decision diagram in [16]) composed by nodes representing the constraints in each filter. The structure is traversed in a second phase of the algorithm, by matching the notification against each constraint. A notification matches a filter when the data structure is completely traversed by it.

2.4.3 Subscription Assignment and Routing

In the above description of the Notification Service, we stated that one of its functions is to stores all the subscriptions issued by subscribers. When considering a distributed implementation, this task is actually accomplished by one or more processes in the brokers set. For the scalability sake, it is a desirable property not to maintain a copy of the whole subscription set on every single broker, but rather sharing the load of storing and managing (i.e. matching) subscriptions among the set of brokers. This allows to effectively exploit the distribution of the Notification Service: in particular, by avoiding subscriptions to be replicated throughout the whole Notification Service, allows the system to tackle scalability issues such as the high traffic generated by the diffusion of each subscription change, the high memory consumption at each broker to store subscriptions and also the high computational power required to match subscriptions[6].

Then, a criterion should be defined in order to choose for each subscription a corresponding target broker to which it is assigned to. Clearly, there is a strict relationship between this problem and the other ones presented in previous Section: the assignment policy must be also considered *(i)* when routing subscriptions to the corresponding target broker and *(ii)* for identifying (and reaching) all possible target brokers when a notification is published.

Given the set of all possible subscriptions SC, we can abstract the *subscription assignment* problem by considering the following function:

$$assign : SC \rightarrow 2^{\Delta} \tag{2.1}$$

The meaning of $assign(\sigma) = \{b_1, .., b_k\}$ is the following: a broker $b_i \in assign(\sigma)$ is the target broker for the subscriber that issued subscription σ. Then, it hosts σ and matches all notifications related to σ. Given a notification e, the task of deriving the target broker set for e, namely *event resolving*, is abstracted by the following function:

$$resolve : \Omega \rightarrow 2^{\Delta} \tag{2.2}$$

Obviously, assignment and resolving are strictly related since a common criteria (*Assignment policy*) must be used to implement the *assign* and *resolve* functions.

It is not straightforward to realize an assignment policy that can easily be implemented both for assignment and for resolving. Subscription assignment strategies used in actual pub/sub systems can be summarized in the following two dual policies:

❒ Access-Driven Assignment (ADA): a subscription σ is stored in the access points for the subscriber $\sigma.s$. In this case access points and target brokers coincides for s. In other words, the value of *assign* depends only on $\sigma.s$, not considering $\sigma.f$.

❒ Filter-Driven Assignment (FDA): subscription filters are partitioned into a set of clusters. Each subscription is assigned to a different broker depending on the cluster it is assigned to. Cluster are determined according

[6]Let us recall that we assume scenarios where a large number of subscriptions must be handled by the system.

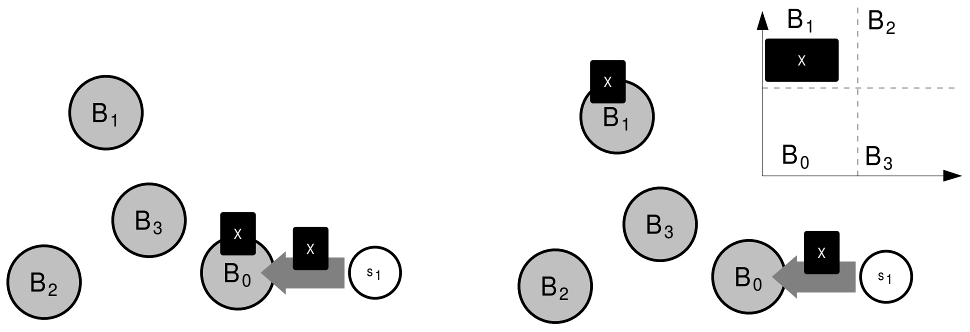


Figure 2.4: Subscription Assignment Policies

to subscriptions' filters. That is, *assign* is the exact dual of the previous case, considering only $\sigma.f$ independently from $\sigma.s$.

Figure 2.4 shows the two different approaches applied to an example subscription X, defined over a 2-dimensional space. On the left, it is shown a ADA approach: subscription is hosted at broker B_0 that has been contacted by the subscriber s_1. On the right, a FDA approach is shown, for the same broker configuration. The plot on the right is a graphical depiction of a simple way to implement the *assign* function: a 2-dimensional space is divided in zones, each one assigned to a broker. Figure shows the graphical representation of the space on a 2-D plot. In this representation, notifications are points and subscriptions are rectangles. X falls in the zone assigned to broker B_1, then X is hosted in B_1.

ADA is the approach commonly used in the majority of distributed pub/sub systems (for example SIENA and Gryphon). No subscription routing is needed because subscriptions are held by brokers directly connected to the subscribers. However, for a correct dispatching of notifications, some routing information has also to be diffused through the network, as we will explain below.

The FDA approach has been formalized in [110], (under the name *subscription partitioning*) and recently, many systems appeared following such a scheme (Scribe [22], Bayeux [117], Hermes [83]). The FDA approach is motivated by the fact that a controlled subscription distribution can allow to better load balance subscription storage and management: all subscription matching the same notifications will be hosted by the same broker, avoiding a redundant matching to be performed in several different brokers. Also notification routing is simplified, consisting in the creation of single-rooted diffusion trees starting from target brokers and spanning all subscribers.

On the other hand, the FDA approach requires all brokers to be aware of the assignment policy used. This problem has no obvious solution in the general case where new brokers may join the system, because the assignment policy can be difficult to adapt to a dynamic scenario: for example consider a policy where the notification schema is partitioned and each partition is assigned to a broker. When a new broker joins, a new partition must be created and all brokers have to be kept aware of the update in the policy. Finding a scalable way to perform this update is not straightforward. [110] proposes two different solutions to the assignment problem, relying on simplifying assumptions: one considers a subscription language comprising only equality constraints, while another allows a general content-based language but assuming that the number of brokers does not change throughout the system lifetime. In Section 2.5 we present assignment policies implemented in systems such as SCRIBE and Hermes.

Let us finally point out that the problem of finding a assignment policy in FDA is equivalent to the problem of determining multicast groups in content-based multicasting, addressed in [91]. Both problems imply a clustering of the subscription set, where in FDA each cluster of subscriptions is assigned to a broker, while in content-based multicasting it is assigned to a multicast group. We stress the fact that none of the aforementioned papers solve the problem in presence of a number of brokers that can change over time.

2.4.4 Event and Notification Routing

We define another function that will be used for stating the event and notification routing problems:

$$avail\ :\ \Omega \times B \times T\ \rightarrow\ boolean$$

The meaning of *avail* relates to the fact that in a distributed system not all processes may have the same view of published information items, i.e. e is not *available* to all processes at the same time [5]. *avail* returns TRUE at a time t when a broker has received an event at t and is able to process it (match, forward or deliver to subscriber).

Event routing means, considering an event e issued by a publisher at a time τ, reaching eventually all the brokers that *can* host subscriptions matching e. Formally, this is expressed as:

$$\forall \sigma : e \sqsubset \sigma, \forall b \in assign(\sigma) \rightarrow \exists t > \tau : avail(e, b, t) = TRUE$$

On the other hand, notification routing means eventually reaching all the brokers that are access points for subscribers that have been identified as interested in the notification. Formally, this is expressed as:

$$\forall e, \forall \sigma : e \sqsubset \sigma, \forall b \in AP(\sigma.s) \rightarrow \exists t > \tau : avail(e, b, t) = TRUE$$

Routing Strategies for Access-Driven Assignment. Obviously under an ADA approach, event routing and notification routing coincide as the target broker for a subscription σ is the access point for its subscriber $\sigma.s$. Thus, when an event is published, the difficult task is getting to know all the brokers that may be subscribers for it, i.e. implementing the *resolve* function. The trivial solution is to broadcast each notification to all the brokers (flooding). This obviously leads to an undesirable waste of network resources, because all notifications are always sent to all brokers even if they do not match any of their subscribers. But avoiding that all notifications are blindly flooded to not-interested subscribers, requires more information on the publisher side. That is, copies of all the subscriptions have to be diffused towards all possible publishers, and in the general case when all brokers may host a publisher for any subscription, this means flooding all subscriptions.

Aiding event routing through subscription redundancy creates a trade-off between notification flooding and subscription flooding. The more brokers are aware of all subscriptions, the earlier notifications that do not match any subscribers can be filtered out. The fact that subscriptions are generally supposed to change at a lower rate than event publication may suggest that the cost of flooding might be worth being paid. However, both simulations studies ([76, 20]) and practical experiences report that subscription flooding can rarely be considered a feasible solution. For example, the complete flooding of subscriptions was a characterizing feature (referred to as "quenching") of an older version of Elvin [99]. Authors in a successive version ([100]) had removed the feature, as it proved to be very costly. A more sophisticated solution for limiting subscription diffusion is the one included in SIENA, and successively refined in Rebeca, that we will present in Section 2.5.

Routing Strategies for Filter-Driven Assignment. Under a Filter-Driven Assignment policy, event routing and notification routing are two distinct phases: in the first one the target broker is reached by the event, in the second one the notification is delivered to all matching subscribers.

Differently from the ADA approach, in FDA brokers that have to be reached by a notification in both phases can be determined in advance: the target broker with the *resolve* function and the matching subscribers' access points from the matching operation itself[7]. Then, event and notification routing reduces to routing to some specific brokers (*Direct routing*). This can be easily implemented either at application-level, or through well-known network routing algorithms (such as link state or distance vector), or by exploiting the direct routing capabilities of an overlay network infrastructure.

[7]Let us recall that clients are excluded from our model. They can be completely replaced by their access points.

Another possible approach for notification routing, proposed in [110], relies on an external centralized notification routing service that receives notifications from brokers and dispatch them to proper subscribers. In this situation, brokers maintain subscriptions and perform matching and the notification service maintains all the references to subscribers.

2.4.5 Classification Framework

All the general solutions related to the various aspect of a Notification Service presented in previous Section can be positioned in the classification framework that we define in the following. The result of the classification is summarized in Table 2.1. The table reports on columns the possible assignment criteria (all brokers, ADA, FDA) and on rows the phases related to notification diffusion. Each cell in the table represents, in each assignment scenario, if a phase is required, how it can be solved and when matching is performed. It is important to point out that here we give a general overview that does not consider the specific optimizations carried out in actual systems. These are presented in the following Section, specifying how they are related to the results in the table.

First, we analyze the case when no particular subscription assignment policy is considered, that means that all subscriptions are propagated to all the brokers in the Notification Service (i.e. the target broker set coincides with the whole broker set). In other words, subscription routing is performed through a broadcast of all subscriptions. Matching can be performed by the broker that receives the publication (Match-first approach), then no event routing is required but only a notification routing, that reduces to a direct multicast to all recipients. The drawback is the high subscription redundancy that implies a high memory consumption and high subscription traffic.

For what concerns Access-Driven Assignment, in its basic realization subscriptions are not replicated but they are retained by access points. This makes necessary for event routing a broadcast of all events through the entire network. Matching is then performed at each access point (Diffusion-first approach).

In Filter-Driven Assignment, we assume that each broker can calculate both the *assign* and *resolve* functions upon publication and subscription, in order to identify the target broker, that can hold a subscription or match an event (Match-at-Target approach). Once the target broker is identified, it can be reached through direct routing from any broker. A notification routing phase is also required after the matching, but this can also be performed in a direct way. The drawback is the additional notification routing phase required. Moreover, implementing a consistent assignment criteria may be no trivial job.

	All Brokers	ADA	FDA
Subscription Routing	Broadcast	NO	Direct
Event Routing	NO	broadcast	Direct
Notification Routing	Direct	NO	Direct
Matching	Match-first	Diffusion-first	Match at Target

Table 2.1: Overview of pub/sub mechanisms

2.5 Surveying Publish/Subscribe Systems

In this Section we specifically deal with real implementations of pub/sub systems. We take into account in details the most popular pub/sub systems, in particular by specifying their characterizing features with respect to the general solutions presented above. Each system will be positioned, according to the classification framework defined above.

2.5.1 TIB/RV

TIB/RV [77] has been one of the first commercial systems to implement the publish/subscribe paradigm. TIB/RV is a topic-based system that relies on the abstraction of an event channel ideally connecting all subscribers interested in a same topic.

In TIB/RV, brokers are structured in a two-level hierarchical architecture. Brokers at the lowest level of the hierarchy are called *rendezvous daemon*. Each network host on which a publisher or a subscriber reside has to run a daemon. Subscriptions are assigned at this level following an ADA approach: each daemon holds the subscriptions for the host on which it runs. Events are diffused through network-level broadcast.

Event diffusion spanning over a WAN is realized through the other type of brokers, namely *rendezvous router daemon*, constituting the higher level of broker hierarchy. Each local network is represented at wide-area level by a single router daemon, that receives all the events directed from the local to other networks and multicasts in the local network notifications received from other networks. Router daemons form an application-level network, in which daemons are connected in couples through TCP connections. Event routing is realized by building multicast trees among router daemons. Each daemon maintains a tree for each subject. A router daemon is added to a tree if there exists at least a subscriber for that subject in the local network represented by that daemon. Since, in order to build trees, daemons have to know both the entire network topology and the current subscription configuration, we can classify the mechanisms used for wide-area diffusion in TIB/RV as a no-assignment policy.

2.5.2 Scribe

An alternative approach for topic-based systems is the one proposed in *Scribe* [22], a research system designed at Microsoft Research. Scribe is built upon an overlay network infrastructure called *Pastry* [95], that allows to perform an efficient large-scale routing of messages in a application-level network of brokers[8]. Each broker in Pastry is assigned an unique identifier in the network and messages can be routed to a specific broker by simply specifying its identifier. Scribe is actually an application written using Pastry and represents a pub/sub interface for it. Subscription routing, event routing and notification routing are implemented in Scribe by leveraging Pastry's direct routing capabilities.

Scribe is a nice example of a system adopting a pure FDA approach. In the following we explain how the assignment policy is implemented. The basic idea is that each topic is assigned a random identifier and the Pastry node with the identifier closest to the topic becomes the target broker for that topic. A multicast tree is built for each topic, rooted at the corresponding target broker. When a new node subscribes for a subject, its subscription is routed by Pastry to the corresponding target broker, that updates the tree structure in order to include the new subscriber. When an event is published for a subject, event routing is performed through Pastry, by directly routing the event to the target broker for that subject. The target broker is simply addressed by the subject's identifier. When an event arrives at the target broker, matching reduces to identifying the correct multicast tree and notification routing is performed by diffusing the notification through such tree.

2.5.3 Gryphon

Gryphon is a content-based system, developed at the IBM Watson research center. Besides being a real implemented system, Gryphon represents the reference framework for all the research in content-based pub/sub carried out at IBM Watson. Relevant research results include a matching algorithm with sub-linear complexity [1], an efficient event routing protocol performing partial matching at each broker [7], and a routing protocol that satisfies exactly-once message delivery [9]. Not all the results in these papers are implemented in the actual system (see the Gryphon web site for details [53]). However, they represent important steps in the evolution of the research in pub/sub systems, then we base our analysis and classification of the Gryphon system upon these papers.

The idea behind the content-based multicast algorithm presented in [7] is

[8]Another topic-based system with a similar approach is Bayeux [117], built over the overlay network infrastructure named Tapestry [116]

to realize a distributed version of the matching algorithm presented in [1]. It is basically a testing network algorithm realized using a tree data structure. In the distributed version, the tree spans all the brokers and the matching of a single constraint is performed at each routing step. This original and effective idea allows to obtain very good performance results. However, the algorithm relies on a very strong assumption: in order to build and keep updated the diffusion tree, each subscription update occurring at a broker must be reflected in the whole system. As we already pointed out, subscription flooding can be very harmful for the overall performance of the system.

A revised version of the Gryphon multicast algorithm is presented in [9]. Here the focus is on addressing reliable delivery of subscriptions, by realizing a fault-tolerant broker architecture. Brokers are organized in a tree structure, rooted at publishers and with subscribers on the leafs. Reliability is addressed in two directions: subscriptions are partially replicated to manage faults of access points, while during event diffusion each broker keeps track of message loss.

Finally, we cite the work of another part of the Gryphon team, devoted to the research in the application of network-level multicast. Relevant results have been produced also in this area, presented in [78, 91, 92], that we already commented above.

2.5.4 SIENA

Another important contribution to the research in content-based pub/sub is the SIENA system [102]. SIENA focuses on providing efficient and scalable notification routing over a wide-area network. Brokers communicate exclusively through application-level connections without exploiting either network-level multicast or overlay network infrastructures. The assignment approach followed by SIENA is ADA. Then, lacking an explicit addressing mechanism for brokers such as in Scribe and not using multicast for event diffusion, one main focus of SIENA event routing algorithm is to avoid flooding events blindly over the entire network.

The idea behind SIENA's routing algorithms is to build logical paths for events from all possible publishers to all subscribers. Paths are built with a subscription routing process, that for each subscription present in the system creates a diffusion tree spanning all brokers, so that each broker knows in which direction it has to route the event in order to reach matching subscribers. This mechanisms allows to prune from the event routing process all the parts of the broker's network that does not contain subscriptions matching that event. Subscription propagation is constrained by exploiting a containment relationships: informally, a subscription σ_1 contains another subscription σ_2 if all events matching σ_2 also match σ_1. When a subscription update occurs, the new subscription is not propagated by a broker if this broker has already propagated a containing subscription before. Another technique to aid the event routing process introduced in SIENA is the advertisements. An advertisement is issued by a publisher to declare the set of events it is going to produce. Advertisements are also considered in building routing paths, to further reduce the set of involved brokers.

With respect to a pure ADA approach, SIENA algorithm increases subscription redundancy in order to create efficient routing paths for events, but a complete replication is necessary only for most general subscriptions.

The SIENA algorithms have become a reference solution for the problem of routing content-based events and subscriptions in an application-level network (*content-based routing* problem). In [76], a general theoretical framework for content-based routing is proposed as well as some variants over the original SIENA algorithm and a performance evaluation.

2.5.5 Hermes

Hermes [83], one of the most recent proposals in the pub/sub research, it is presented as a pub/sub middleware rather than a simple system, because it encompasses also other functions such as type checking and security. In the context of our presentation, Hermes gathers in an interesting way several ideas from the systems described above.

Hermes is a system based on a FDA approach, with a type-based subscription model. Differently from Scribe, it is implemented as a network of brokers rather than exploiting an overlay network infrastructure[9]. The FDA policy is realized considering the type of a subscription: a subscription is assigned to a broker whose identifier matches the hash of the type name. The FDA policy organizes subscriptions in coarse-grained clusters (types) and it may happen that the system contains more brokers than types, leaving some brokers not assigned to any type. At the same time, differently from SIENA, only a single copy of each subscription is retained in the system, at the corresponding target broker.

Event and notification routing are implemented with SIENA-like algorithms. Paths are created for routing events belonging to a specific type to the corresponding target broker. Notification routing follows the same idea, with a diffusion tree for each type, rooted at the target broker and having each subscriber as a leaf. Content-based filtering of subscription is performed directly at recipients. Thus in Hermes there is only one spanning tree for each

[9]In a recent paper ([84]), authors described a overlay network implementation of Hermes. However, the nature of the routing algorithms does not change with respect to the original version described here and the overlay network infrastructure is used only as a communication facility.

type, whereas in SIENA subscription routing builds in practice a spanning tree from each possible publisher to all subscribers.

2.6 Concluding Remarks

Publish/subscribe is now widely recognized as a hot research topics, inspiring several areas such as databases, distributed systems or software engineering. In this Chapter we gave an overview of the pub/sub research area by classifying the several research problems that it hides, the relationships among them and the solutions that have been proposed in the literature. At the best of our knowledge this is the first attempt to provide a general survey of the area that tries to go beyond a simple coarse-grained classification [71]. In particular, the different routing schemes have been classified considering not only the typical "topic-based vs. content-based" distinction, but accounting also for novel important solutions such as the leveraging of peer-to-peer overlay networks and the routing based on a filter-driven assignment policy. Subscription assignment itslef is a novel concept introduced here for the first time. Anyway, several important research systems [30, 99, 52] have been cut out of the presentation, but the idea was to select the systems implementing the most representative solutions, trying to highlight the most important and peculiar ideas in the field.

Chapter 3

Modelling Publish/Subscribe Systems

In Chapter 2 we surveyed the large amount of work done in the field of publish/subscribe systems focusing on scalability, efficient information delivery or efficient and expressive information matching. On the other hand, only one specific contribution exists [76] giving an unambiguous definition of the computational model underlying a pub/sub system. This step is necessary for carrying out, for example, an analytical study of the performance of a pub/sub system, which is the base of a rigorous QoS policy. The lack of this rigorous approach is currently one of the main pitfalls of pub/sub which limits its applicability, for example, to mission critical systems.

In this Chapter we propose a computational model of a pub/sub Notification Service, where the latter is abstracted as a black box connecting all participants to the computation. The operations done by this box (i.e., subscription/unsubscription storage and publication diffusion) are modelled by two delays, namely the subscription delay and the diffusion delay, which characterize, respectively (i) the non-atomicity of the subscription/unsubscription storage and (ii) the non-instantaneous diffusion of a notification. These delays depend of course on the implementation of the Notification Service (e.g. centralized, network of brokers, etc.). This model produces a global history of the computation, on which we give safety and liveness properties.

In distributed computing, safety properties express the constraint that "something bad will not happen" in the system. In the case of a pub/sub system this simply means that all notifications must be previously published and no spurious notifications has to be delivered. For what concern liveness, this usually mean "something good eventually happens". However, in this dynamic context where (i) participants can subscribe and unsubscribe dynamically and (ii) there are operations which take time to take effect, this sentence should be reworded as follows "under some timing conditions something good happens". Then we propose a liveness property which states when a notification belongs to the history: this is affected by the interval a subscription is "active" and by the two delays which act as a filter for the generation of the notification after the execution of a corresponding publication.

Based on the computational model, we provide a probabilistic model for measuring the effectiveness of a Notification Service in notifying publications to the set of the interested subscribers. More specifically, we evaluate the probability d that a publication x issued at time t will be notified to each subscriber matching x, provided that the subscription was active at t. Therefore, the system behaves ideally if this probability is equal to 1. We study this probability as a function of the subscription delay and of the diffusion delay.

Even though the granularity of this model is quite coarse, we believe that it can be very useful for the designer of a Notification Service, that can predict the probability of delivering notifications, only by estimating the two delays. A simulation study, carried out on a real pub/sub implementation, validates the analytical model.

The chapter is structured as follows: Section 3.1 introduces the formal framework, Section 3.2 presents the analytical probabilistic model for performance evaluation, Section 3.3 presents the experimental results and the comparison with the analytical ones. Finally, Section 3.4 surveys the related work.

3.1 A Framework for Publish/Subscribe

We consider a distributed system composed of a set of processes $\Pi = \{p_1, \dots, p_n\}$ that communicate by exchanging information in a publish/subscribe communication system. Processes are decoupled in the sense that they never communicate directly within each other but only through a common *Notification Service* (NS). Processes can act both as producers and consumers of information, taking on the role of *publishers* and *subscribers*, respectively. The concepts of subscriptions, notification and matching follows from the definitions in Section 2.1 of Chapter 2.

3.1.1 Process-NS Interaction

In the following, we formalize the interaction model sketched in Section 2.1. The execution of a publish/subscribe system comprises both process-side operations, started by subscribers and publishers, and NS-side operations, started by the NS. More specifically, any process p_i would be able to register (and cancel) a subscription or to publish a notification in the system, but it is actually the NS that has the role of notifying a matching occurrence to interested subscribers.

We denote as $op = \{sub(\sigma), usub(\sigma), pub(x), ntfy(x)\}$ respectively the operations of registration of a subscription σ, cancellation of a subscription σ, publication of a notification x and issue of the notification of x.

Then, the operations $sub(\sigma), usub(\sigma), pub(x)$ are issued by a process and executed by the NS, while $ntfy(x)$ is issued by the NS on a process p_i and then executed by p_i. The $ntfy(x)$ issue occurs after (i) the $pub(x)$ execution and (ii) a matching operation executed within the NS. Note that the NS issues $ntfy(x)$ on the set of processes computed after the matching operation.

3.1.2 Computational Model

To simplify the presentation, we assume the existence of a discrete global clock whose range $\mathcal{T}$ is the set of natural numbers. We stress the fact that this is only a fictional, abstract device to which the processes *do not have access*. We will use it only for convenience of specification.

The first modelling step is the representation of the execution of each process. Through an abstract representation of the processes' computation we describe which global computations are allowed in a NS, by specifying properties that characterize them.

We assume either the *issue* of an operation $op = \{pub(x), sub(\sigma), usub(\sigma)\}$ at time t at a process p_i or the *execution* of $op = ntfy(x)$ at p_i at time t produces an *event* $e_i(op, t)$ at process p_i[1]. We denote then the *local history* of a process p_i as the set of events occurred at p_i and ordered by their occurrence time $h_i = \{e_i(op, t_1), e_i(op, t_2), \ldots e_i(op, t_m)\}$ (with $t_1 < t_2 < \ldots < t_m$). The global computation is then the *global history* $H = \langle h_1, h_2, \ldots, h_n \rangle$, i.e. a collection of local histories, one for each process.

Any two successive events $e_i(sub(\sigma), s)$ and $e_i(usub(\sigma), u)$ $(s < u)$, define a *subscription interval* of p_i for the subscription σ, denoted by $I(\sigma)$. Such subscription interval includes all events $e_i(op, t)$ s.t. $s \leq t \leq u$. Therefore, to univocally identify each subscription issued in the system by the same process, a generic subscription σ becomes a triple (ϕ, p, s) where $\sigma.s$ indicates the time in which the subscription is issued. The time between s and u actually represents the time in which the subscription σ is active from the subscriber view-point. We denote such time interval as $T_{ON}(\sigma)$. A subscription interval is defined also by those *sub* events that have no corresponding *usub*. In this case the interval will include all events that occur after the *sub* and T_{ON} will be consequently infinite. Figure 3.1 shows an example of global history of three

[1]In this Chapter we use the term "event" only referring to events belonging to the internal computation of processes. Pieces of information produced by publishers are *always* referred to as "notifications".

processes, with two subscription intervals $I(\sigma)$, $I(\sigma')$ and their corresponding T_{ON}.

Safety properties

Safety properties pose constraints on which global histories are not allowable in a NS. The first property has to state the basic semantics of the system: a subscriber cannot be notified for an information it is not interested in. Formally:

$$\forall\ e_i(ntfy(x),t) \in H \ \Rightarrow\ e_i(ntfy(x),t) \in I(\sigma) \text{ s.t. } x \sqsubset \sigma.\phi \ \wedge\ \sigma.p = i$$

P1: Legality

In Figure 3.1 a generic computation satisfying Legality is shown: supposing that x and y match σ, then both notify events of x and y in p_i fall in the subscription interval $I(\sigma)$ of p_i. While Legality states that a notify event belongs to H only if it is included in a subscription interval matching that event, we need a property that ensures the notify events are not invented by a process. This is taken into account by the Validity property which states as follows:

$$\forall\ e_i(ntfy(x),t) \in H \Rightarrow \exists\ e_j(pub(x),t') \in H \text{ s.t. } t' < t$$

P2: Validity

The computation in Figure 3.1 also respects Validity: then both notify events, $e_i(ntfy(x),t_2)$ and $e_i(ntfy(y),t_4)$ follow the corresponding publications, $e_i(pub(x),t_1)$ and $e_i(pub(x),t_3)$, as $t_1 \leq t_2$ and $t_3 \leq t_4$.

Once safety properties are defined, it is interesting to understand under which condition a notify event should be generated, i.e. to define a Liveness property. As just said, the global history in Figure 3.1 satisfies safety. However supposing x' matches σ', should we expect that the NS system generates a computation with the notify event for x' in $I(\sigma')$? To answer this question is first essential to make some considerations about how a NS is physically built. This is actually the aim of the following Section.

3.1.3 NS Implementation Parameters

Recalling from previous Chapter, we can say that the NS has two main tasks:

❒ store and manage subscriptions from processes caused by the issue of subscribe/unsubscribe operations;

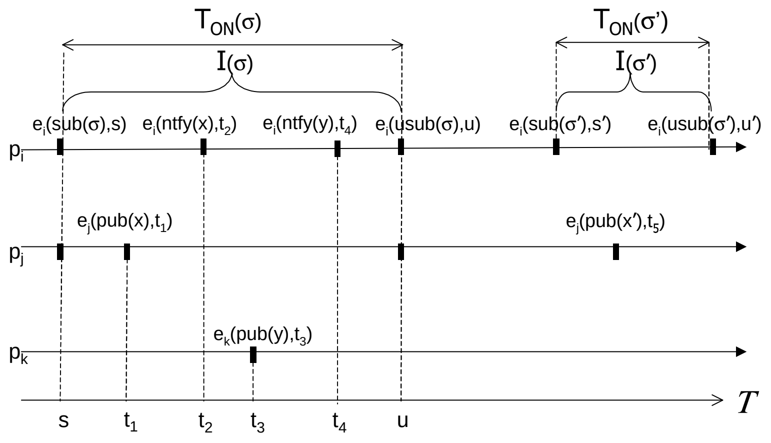


Figure 3.1: Global History respecting Safety

❒ diffuse a notification x to the interested subscribers after a publish operation was issued by a process;

Obviously, behind this abstract and informal description of a NS, there exists an actual NS physical implementation (e.g. centralized, distributed, network of brokers etc.) that performs the desired functionality. In order to capture the behavior of *any* NS implementation we define two parameters that respectively take into account (i) non-instantaneous effects of subscribe/unsubscribe operations and (ii) the non-instantaneous diffusion of a notification x to interested subscribers after a publish operation issued by a process. These parameters model the time required for the internal processing at the NS and the network delay elapsed to route subscriptions and notifications, in a distributed implementation. Let us finally assume that any message sent by a processes of a NS implementation uses reliable channels.

Subscription/unsubscription delays.

When a process issues a subscribe/unsubscribe operation, the NS is not immediately aware of the occurred event. In other words, at an abstract level, the registration (resp. cancellation) of a subscription takes a certain amount of time to be stored into the NS. This time encompasses for example the update of the internal data structures of the NS and the network delay due to the routing of the subscription among all the entities constituting the NS. To consider such non-instantaneous operations, we define a maximum acceptable threshold of time (implementation dependent) after which a subscribe/unsubscribe operation is *surely* stored into the NS. As an example, in a distributed implementation of a NS, this means *each entity* implementing the NS after this threshold of time is aware of the registration/cancellation operation.

We denote such delay as T_{sub} for subscribe operations and as T_{usub} for unsubscribe operations. Therefore if a subscribe operation is issued at time s then it takes effect at a time t such that $s < t \leq s + T_{sub}$ [2]. The same holds for unsubscribe operations, i.e. an unsubscribe operation, issued at time u, takes effect at a time t' such that $u < t' \leq u + T_{usub}$.

To model this effect on the NS, we consider the NS characterized by a *subscription configuration sc* composed by a set of subscriptions. In particular, we define $sc(t) = \{\sigma_1, \sigma_2, ...\sigma_m\}$ the set of all subscriptions stored into the NS at time t. We assume the initial configuration $sc(t_0) = \emptyset$. Therefore if a subscribe (resp. unsubscribe) operation for a subscription σ takes effect at time t (resp. t') then $\sigma \in sc(t)$ (resp. $\sigma \notin sc(t')$). As a consequence, even though t and t' are a-priori unknown, we can state with certainty that $\sigma \in sc(s + T_{sub})$ and $\sigma \notin sc(u + T_{usub})$. For example in Figure 3.2, $\sigma \in sc(t_1)$ and $\sigma \notin sc(t_2)$, but in both $[s, t_1]$, $[u, t_2]$ time intervals there is uncertainty whenever $\sigma \in sc$ or not.

At an abstract level each subscription of the NS state at time $t \in \mathcal{T}$ can therefore be *stable* (i.e., it surely belongs to NS state) or non-stable. A subscription σ is stable with certainty at time t, iff $s + T_{sub} \leq t \leq s + T_{ON(\sigma)}$.

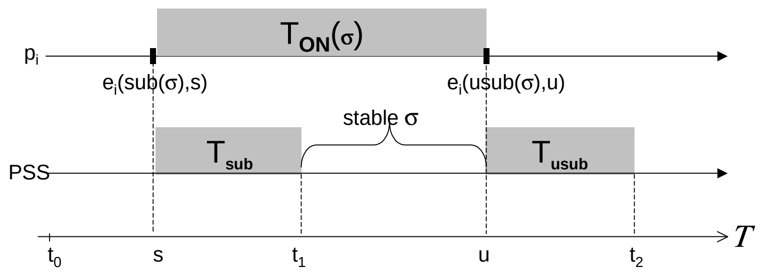


Figure 3.2: Subscription/unsuscription delays

Diffusion delay.

As soon as a publication is issued, the NS performs a *diffusion* of the information: it performs a matching to compute the set of interested subscribers and sends the notification to them. Note that, depending on the NS implementation, the diffusion can be performed in several ways, as described in Section 2.4. Without entering implementation details, we can say that this operation takes a certain amount of time *during* which the NS computes and

[2] In our framework we reasonably assume for each subscriptions σ we have $T_{ON}(\sigma) > T_{sub}$.

issues notify operations to interested subscribers, i.e. diffusion takes a non-zero time. Let us suppose that a publication of a notification x is made at a given time t, and there is a matched subscription σ that is stable at time t, i.e. $\sigma \in sc(t)$. Then the NS starts the diffusion to notify x to $\sigma.p = p_i$. We denote as Δ_i the time elapsed in order to complete the diffusion of x to p_i. An event $e_i(ntfy(x), t')$ can be generated only at time $t' \leq t + \Delta_i$. After the completion of the diffusion, the notification x disappears from the NS, i.e. a further notify event can no longer be generated.

Note that in the worst case scenario, the set of subscribers to be notified and the whole set of processes coincides. In this case the diffusion takes the maximum time among $\{\Delta_1, \Delta_2, \dots \Delta_n\}$. We define such maximum delay as *diffusion delay*, denoted T_{diff}.

To clarify the meaning of the diffusion delay see Figure 3.3. For sake of simplicity and without loss of generality we assume that the communication delay between a process and the NS is zero. This implies that (i) a notification published by a process immediately gets the NS, and (ii) if NS issues a notify operation on a process p_i, the corresponding local event at p_i is immediately generated. Immediately after the publication of the notification x at the time t_1, the NS, during T_{diff}, notifies the interested subscribers. Supposing that $\{p_j, p_k, p_h\}$ is the set of interested subscribers then $T_{diff} = max\{\Delta_j, \Delta_k, \Delta_h\} = \Delta_h$. Let us remark that each generic interested subscriber p_i is notified in specific instant of time $(t+\Delta_i)$ but Δ_i is not a-priori known.

It is important to point out that the set of interested subscribers is clearly computed on the basis of NS's configuration. However, *how and when* the state is considered, is implementation dependent. Moreover, *the configuration can change during the diffusion.* In the following these aspects will be clarified.

3.1.4 Liveness Property

The concept of "interested subscriber" has been considered quite intuitively until this point. The desirable NS behavior is the following: once a notification is published (i.e., $e_j(pub(x), t)$ is generated in H), x is notified to each interested subscriber; but what is an interested subscriber? Ideally it is a process p_i that expresses its interest for x through a subscription σ s.t. $x \sqsubset \sigma.\phi$ and $\sigma.s \leq t \leq \sigma.s + T_{ON}(\sigma)$. However the NS system is surely aware of the subscription σ by p_i only when the subscription becomes stable, i.e. at time $\sigma.s + T_{sub}$. Then, at first check, an interested subscriber seems to be a process whose subscription is stable (i.e. belonging to the NS state), at the moment in which the matching information is published, i.e. $\sigma.s{+}T_{sub} \leq t \leq \sigma.s{+}T_{ON}(\sigma)$.

However as (i) the interest of a subscriber is a dynamic dimension and (ii) a notify can be issued to a subscriber at any time during the diffusion

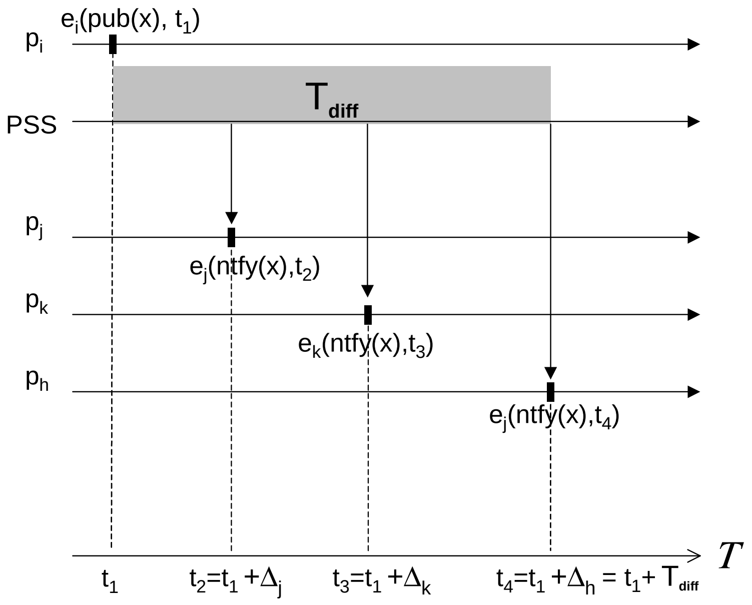


Figure 3.3: Example of Diffusion

interval of the corresponding publication, it is still difficult to characterize the exact behavior of the system. Let us point out this with an example. Let p_i be a process producing a subscription σ and p_j be a process producing an event $e_j(pub(x), t)$ such that $x \sqsubset \sigma.\phi$ and $\sigma.s + T_{sub} \leq t \leq \sigma.s + T_{ON}(\sigma)$. However, if NS *is able to notify* x at p_i *only* at a time $t' = t + \Delta_i$ such that $t' > \sigma.s + T_{ON}(\sigma)$ then p_i will discard x as it is not longer interested to x.

Then, the definition of a liveness property, that states exactly to which subscribers a publication is notified to, must be necessarily defined considering both the subscription/unsubscription delays and the diffusion delay. Given a subscription interval $I(\sigma) \doteq [\sigma.s + T_{sub}, \sigma.s + T_{ON}(\sigma)]$, This property can be stated as follows:

$$\forall(\ e_j(pub(x), t) \wedge (I(\sigma) \in H \text{ s.t. } I(\sigma) \supset [t, t + T_{diff}]) \\ \Rightarrow \exists\ e_{\sigma.p}(ntfy(x), t'') \in H$$

P3: Liveness

This property states that the delivery of a notification can be guaranteed only for those subscribers that maintain their subscriptions stable for the entire time taken by notification diffusion (diffusion delay). In other words, Liveness property defines the NS system condition under which a notify event belongs to the global history. However, a notify event can also belong to the history even though this system condition is not verified. This is due to the uncertainty on the system state and on the diffusion time of an information through the NS, as shown in the example depicted in Figure 3.4.

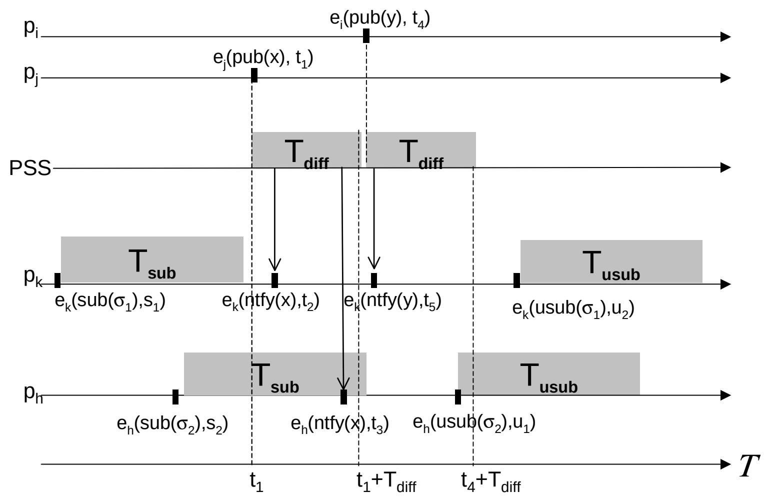


Figure 3.4: Global History with not expected notify events

From application of the Liveness property the only notify events *guaranteed* to be in the global history are $e_k(ntfy(x), t_2)$ and $e_k(ntfy(y), t_5)$, as p_k has a subscription (matched by both x and y) stable during the whole diffusion of x and y. However the global history contains also $e_h(ntfy(x), t_3)$. This depends on the fact that (i) the subscribe operation for subscription σ_2 has taken effect *before* the $\sigma_2.s + T_{sub}$, (ii) NS has made the diffusion relying on a state containing σ_2, and (iii) the diffusion to p_h has completed before $t_1 + T_{diff}$. Note that such "lucky" conditions *may* occur but the probability of its occurrence is not equal to one.

3.1.5 Persistent Notifications

The liveness property introduced above, can be extended by considering the possibility for the NS to persistently store notifications for a finite, non-zero amount of time. Persistence is usually exploited in distributed NS implementations to provide reliable delivery of notifications to processes: periodically processes request each other the list of notifications they sent and if a process could not find some notification it requests their retransmission.

However, persistence can strongly influence the semantics of delivery if we assume that while the notification is maintained in the NS it can be delivered to processes that subscribe *after* it has been published. In particular, let us introduce a persistence interval $T_{ON}(x)$, representing the time interval a notification x is stored in the NS after it has been published by a process. If x is published at time t_x, all subscribers whose subscription becomes stable after t_x have to be notified. This behavior is expresses by the following extended definition of the liveness property:

$$\begin{gathered}\forall(\ e_j(pub(x),t) \wedge (I(\sigma) \in H \text{ s.t.} \\ (I(\sigma) \supset [t, t+T_{diff}] \vee I(\sigma) \cap [t+T_{diff}, t+T_{ON}(x)] \neq \emptyset) \\ \Rightarrow \exists\ e_{\sigma.p}(ntfy(x), t'') \in H\end{gathered}$$

P3a: Liveness (with persistent notifications)

Differently from the previous definition of Liveness, in this case for the issue of a notification x to be guaranteed to a process p with a matching subscription σ, the subscription has to be maintained for an interval that (i) entirely contains the diffusion interval of x (as in P3) or (ii) intersects the interval during which x is persistently stored in the system.

As pointed out above, depending on the value of $T_{ON}(x)$, the semantics of the NS can be radically altered. In particular, we can consider three representative cases:

0-persistence : each published notification x expires as soon as it becomes available. Only processes whose subscription matches x at the very moment of its publication are guaranteed for notification. This implementation scheme is very low demanding since it does not require storing notifications. However, it is subject to runs between concurrent *pub* and *sub* events, i.e. a subscriber may miss a notification x if its subscription is even slightly delayed with respect to the publication.

Δ-persistence : each published notification x expires after $\Delta > 0$ from the instant it becomes available. Garbage collection is performed by the pub/sub system on all expired notifications. This is more resilient to runs between the publisher and subscribers: a subscriber whose subscription "is seen" in the system after Δ since x is available, can be still satisfied. On the other hand high values for Δ may provoke undesirable out-of-date notifications.

∞-persistence : each published notification x remains available in the pub/sub system for an indefinitely long time. When a subscriber installs a new subscription σ it will receive all the previously published and already available notification that match σ. This implementation style obviously requires an ideal infinite memory to store all the notifications, since no notification is ever garbage-collected.

This classification can be related to the non-deterministic behavior deriving from the time decoupling between participants, that is one peculiar features of the pub/sub paradigm. In general, the more an information item remains available in the system, the less non-determinism is experienced (for example, the effect of runs between publications and subscriptions is limited). Reduction of non-determinism increases the probability that an intended receiver gets the information. If the information is stored in a persistent way, non-determinism is completely removed and this probability goes to one [107]. Of course, this impacts on the size of the memory necessary within the system.

Then, we can say that the three classes feature decreasing levels of non-determinism on the other hand requiring increasing memory. However, we point out that this is not the case where a single class can be identified in absolute as better than the others, whereas each class is suited to meet different application requirements. Hence we give examples of applications that may belong to each class.

An example of a 0-persistent application is a stock exchange system. Notifications represent instant values of stock quotes, expiring very quickly. Since the publication rate is high, missing notifications due to runs poses no problem, since subscribers can get a new information after a short time.

An example of a Δ-persistent application is a daily news diffusion system. Each information item represents the daily issue of the news, having a lifetime of 1 day. Suppose issues are published every morning at 3 A.M., an issue will be notified also to processes submitting their subscription during the day.

An example of an application based on ∞-available information is a digital library where catalog updates are published as information items, kept available for future subscribers. A new subscriber will receive all the previous notifications in order to build its local copy of the catalog.

3.1.6 On the liveness specification in dynamic systems

As we pointed out in [6], understanding and comparing different publish/subscribe systems is quite a difficult task due to informal and different semantics. From this stems the requirement of precisely defining formal semantics in terms of safety and liveness properties as in any distributed system.

To our knowledge the first step in this direction was done in [76], where the author defines safety properties which are actually similar to the ones defined in Section 3.1.2. However defining "no bad thing can happen" is the easy part of the job in dynamic distributed systems (such as publish/subscribe applications). The tricky part is defining a property of progress of the whole system (i.e., the liveness property) when processes behave independently and dynamically. In the classical (static) distributed system, liveness constrains a system to *eventually* make progress on the global computation towards a certain target. [76] defines liveness along this line. *"If a notification matching a set of active subscribers is published, then each subscriber will eventually be notified unless it cancels its subscription"*. In other words if a subscriber *never disconnects* with a subscription matched by a published notification, it eventually will be notified for that notification. Then nothing is guaranteed if the subscriber remains connected only for a certain time (even though this is a very long time!). Of course the assumption that a subscriber never disconnects is unrealistic in a pub/sub system.

Another example of the inadequacy of the liveness property as defined in classical distributed systems comes from the crash-prone model. In this setting, the verification of the liveness property ensures progress toward the termination of a computation. This usually requires the assumption on a minimum number of correct processes in the system (i.e., processes that never fail). If we make a parallel with a pub/sub system, this means to make an assumption on the minimum number of subscribers that never disconnects. It is clear that in pub/sub such an assumption does not make any sense[3]. A disconnected process *is not a bad process* as "disconnection" is a matter of life in a pub/sub and not an undesirable event to cope with. A liveness specification for this dynamic context should capture this normal behavior.

Roughly speaking, our liveness definition actually considers "each notify event" as the target of our computation and defines timing assumptions under which a notify event is in the global history of the computation (i.e., this event has to be notified by the NS). This presence depends of three delays ($T_{diff}, T_{sub}, T_{usub}$) which abstract the dynamic behavior of the computation and the NS implementation. Thanks to the latter point our Liveness condition can also be used to compare different NS implementations. To explain this point, consider the following example. Suppose to have two different NSs managing the same set of clients and the same type of subscriptions and notifications. Moreover suppose that:

1. a process publishes in both systems a notification matching an active subscription made by the same subscriber (a subscriber connected to both systems),

2. the subscription will remain active for two days after the publication but will be notified only by the first system.

In this scenario both systems satisfies liveness as defined in section [76], but are they equally good? It seems that the latter is a "lazy" system, while the former is more reactive and effective. In the next section we show how

[3]Defining a liveness that gives guarantees if and only if a process never disconnect is the equivalent of not giving any guarantee.

this reactiveness can be measured to give an idea of the effectiveness of the implementation.

Let us finally remark that if T_{diff} was infinite our definition of Liveness would not guarantee anything as the classical Liveness stated in [76]. However, differently from [76] when the three delays are finite (typical practical case) some subscription (satisfying conditions stated in the Liveness property) must be notified.

3.2 Analytical Model

The semantics identified through the abstract computational model allows us to reason on the practical consequences of T_{sub} and T_{diff}. Starting from the consideration that high values of the delay can prevent some subscribers from receiving notifications that were issued during the publication time, the idea is to take the probability that this could happen as a general performance parameter (namely the *notification loss*) for the NS.

In this Section we provide a simple and general analytical model for the computation of the notification loss, given the implementation parameters T_{sub}, T_{diff}. We also specialize the model discussing how the practical deployment of a pub/sub system can influence such parameters and, subsequently, the notification loss. In particular, we identify three *operational parameters* that characterize the size of the system and the speed of the propagation of subscription and notifications. These parameters can be measured experimentally, subsequently obtaining T_{sub} and T_{diff}, hence the notification loss.

3.2.1 Measuring Notification Loss

Let x be a notification issued at time t and p be a generic process that has a subscription at time t matching x. We denote as d the probability that x is notified to p (*notification probability*). Therefore, the *notification loss* is the probability that x is not notified to p (i.e., $1 - d$), though p has a matching subscription.

Our analysis rests on the following assumptions:

1. the process p issues the subscription σ for a period $T_{ON} \geq T_{sub} + T_{diff}$;
2. any other subscription can only be issued by p after T_{usub} from the last *usub* operation
3. the time a publication matching σ is issued is a uniformly distributed random variable defined over the subscription interval T_{ON};
4. all publishers have the same probability to generate a notification matching σ.

5. each notification is persistently stored in the NS for a time $T_{ON}(x)$

The delay T_{usub} does not affect d because we assumed that a subscriber can issue a new subscription only after T_{usub} and, by definition, σ has been cancelled from the NS's configuration after this time interval.

In the following we give two expressions for d respectively considering $T_{ON}(x) = 0$ (volatile notifications) and $T_{ON}(x) > 0$ (persistent notifications).

Volatile notifications. Let t_{sub} be the time the process p issues the *sub* operation, t_{usub} the time when p issues the corresponding *usub* operation and t_{pub} the time the notification x is published. Then, the NS guarantees the delivery of any publication occurring at a time t_{pub} such that $t_{sub} + T_{sub} \leq t_{pub} \leq t_{usub} - T_{diff}$. This means, in fact, that the publication was issued when the subscription was stable and there was enough time for information diffusion to be completed. Moreover, for those publications such that $t_{sub} < t_{pub} < t_{sub} + T_{sub}$ as well as for those with $t_{usub} - T_{diff} < t_{pub} < t_{usub}$, there is also some probability for being notified.

For example, let us consider a distributed implementation of the NS as a network of brokers. Then, roughly speaking, it is possible that x was published by some process "close" to p so that, after a delay $t < T_{sub}$, the portion of the NS involved in the diffusion of x towards p has already received the updates for correctly notifying x, as suggested by our simulations.

To model this aspect, we denote with $f(t)$ the probability density function that the NS notifies x to p, given that x was issued at time $t_{pub} = t_{sub} + \tau$, where $0 \leq \tau \leq T_{sub}$. Clearly, $f(t)$ must be a monotonically increasing function with $f(0) = 0$[4] and $f(T_{sub}) = 1$. In our experiments, $f(t)$ corresponds to the function plotted in Figure 3.10(a).

Also, $g(t)$, where $0 \leq \tau \leq T_{diff}$, is the probability density function that a notification published at a time $t_{pub} = t_{usub} - T_{diff} + \tau$ is notified to p. In a real implementation, this function captures the probability that the notification x reaches p before it unsubscribes for σ. The function $g(t)$ must be a monotonically decreasing function with $g(0) = 1$ and $g(T_{diff}) = 0$. In our experiments, $g(t)$ corresponds to the function plotted in Figure 3.10(b).

Figure 3.5 sketches the overall probability density function $P(t)$ that a notification x matching σ, issued at a time t inside T_{ON}, is notified by the NS.

Due to the assumption (iv), $\frac{1}{T_{ON}}dt$ is the conditional probability that $t_{pub} \in [t, t + dt]$ $(t_{sub} \leq t \leq t_{usub})$, given that an information was published during the subscription interval. Moreover, $\frac{P(t)}{T_{ON}}dt$ is the probability that the an event is published in the interval $[t, t + dt]$ and it is notified.

[4]Actually, the value is slightly higher than 0, because there is the probability that x is issued by p itself. We discuss this below.

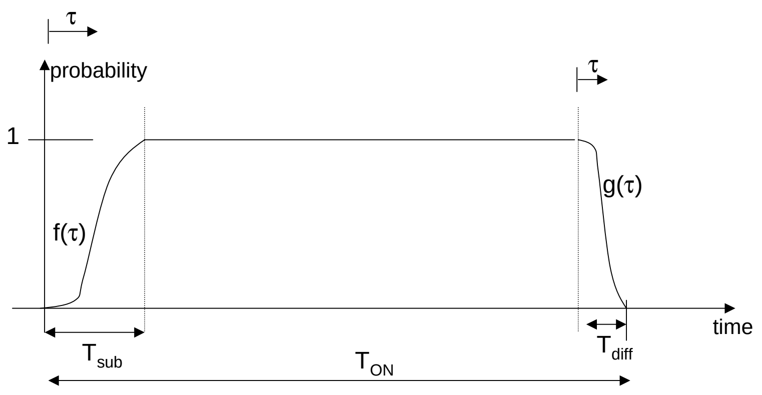


Figure 3.5: A sketch of $P(t)$, volatile notifications

Applying the total probability theorem, we can thus evaluate d as following:

$$d = \frac{1}{T_{ON}} \left(\int_0^{T_{sub}} f(t)\,\mathrm{d}t + \int_{T_{sub}}^{T_{ON}-T_{diff}} 1\,\mathrm{d}t + \int_0^{T_{diff}} g(t)\,\mathrm{d}t \right)$$

that can also be rewritten as

$$d = \frac{T_{ON} - T_{diff} - T_{sub}}{T_{ON}} + \frac{1}{T_{ON}} \left(\int_0^{T_{sub}} f(t)\,\mathrm{d}t + \int_0^{T_{diff}} g(t)\,\mathrm{d}t \right) \qquad (3.1)$$

Note that d is directly proportional to the area of the curve depicted in Figure 3.5. Clearly, if $T_{diff} = T_{sub} = 0$ then the NS behaves as an ideal system with $d = 1$ (all publications are immediately notified).

Persistent notifications. The evaluation of d with persistent notification is basically the same as the one performed above for volatile ones, just considering also the fact that when notifications are stored inside the NS for a time $T_{ON}(x) > 0$[5], a subscriber receive also notifications issued *before* its own matching subscription was stable. More precisely, the NS guarantees the delivery of any notification occurring at a time t_{pub} such that (i) $t_{sub}+T_{sub} \leq t_{pub} \leq t_{usub}-T_{diff}$ (as in the previous case) or (ii) $t_{sub}+T_{sub}-T_{ON}^{x} \leq t_{pub} \leq t_{sub}$. In particular notifications satisfying the the latter condition will be delivered because being the subscription surely stable at $t_{sub}+T_{sub}$, it will get notifications issued up to a time T_{ON}^{x} in the past.

[5]We will use a simplified notation, writing T_{ON}^{x} instead of $T_{ON}(x)$.

For what concerns notifications issued between $t_{sub} - T^x_{ON}$ and $t_{sub} + T_{sub} - T^x_{ON}$, the probability of their delivery is conditioned to the probability that the subscription σ is stable before a time T^x_{ON} in the future. Hence, the trend of $P(t)$ will simply follow that of the function $f(t)$. Finally, persistent notifications does not have any influence on notifications published in proximity of the unsubscribe of σ.

The analytical evaluation of d obviously resembles the one obtained for volatile notifications, but with an added term accounting for the probability of "past" notifications:

$$\begin{aligned} d = \frac{1}{T_{ON} + T^x_{ON}} \Big(\int_0^{T_{sub}} f(t)\,\mathrm{d}t + \int_{T_{sub}}^{T_{sub}+T^x_{ON}} 1\,\mathrm{d}t + \int_0^{T_{sub}} f(t)\,\mathrm{d}t + \\ \int_{T_{sub}}^{T_{ON}-T_{diff}} 1\,\mathrm{d}t + \int_0^{T_{diff}} g(t)\,\mathrm{d}t \Big) = \\ = \frac{1}{T_{ON} + T^x_{ON}} \left(T_{ON} - T_{diff} - 2T_{sub} + T^x_{ON} + 2\int_0^{T_{sub}} f(t)\,\mathrm{d}t + \int_0^{T_{diff}} g(t)\,\mathrm{d}t \right) = \end{aligned}$$

$$= 1 - \frac{2T_{sub} + T_{diff}}{T_{ON} + T^x_{ON}} + \frac{1}{T_{ON} + T^x_{ON}} \left(2\int_0^{T_{sub}} f(t)\,\mathrm{d}t + \int_0^{T_{diff}} g(t)\,\mathrm{d}t \right) \quad (3.2)$$

This expression is valid for $T^x_{ON} > T_{sub}$.

3.2.2 Analytical results

In order to provide an analytical expression for d, the two functions f and g have to be somehow specified. The shape of the curves will obviously be determined by the mechanisms used internally by the NS for routing subscriptions and notifications. These mechanisms are dependent from three operative parameters: N representing the size of the system in terms of the number of entities it is composed of; s and r, respectively the subscription and the notification update rates, representing the efficiency of the routing processes. In the following, we first consider a general distributed implementation for the NS and give an analytical evaluation for the two functions in this case. Then, we investigate how the operative parameters of the NS are related to the implementation parameters T_{sub} and T_{diff}.

NS Reference Model

As we pointed out in Chapter 2, the assignment policy strongly influences the algorithms for routing notifications and subscriptions. The following analysis is carried out assuming a NS with volatile notifications and Access-Driven

Assignment policy, that is all subscriptions are retained at the access point for their subscriber.

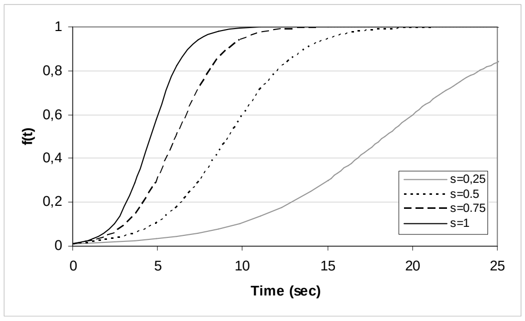
We consider a general case where a *subscription routing* algorithm is used to make all participants in the system aware of each active subscription. Thus, a subscription is stable after its routing process is terminated. That is, the subscription delay is the *maximum* convergence time of the subscription routing algorithm. The f function should represent the variation over the time of the number of NS participants that received a subscription update. The maximum convergence time is obtained when all subscriptions are flooded within the entire system, travelling all its participants.

Once the subscription is stable, the corresponding subscriber can be contacted when a matching notification is published. A *notification routing* algorithm is then used to propagate the notification among brokers in order to reach all the subscribers that the subscription matching the such a notification. During the routing process, a broker receiving a notification from another broker has the opportunity to issue the notify operation to its local matching subscribers. Moreover, it can forward the notification to the other brokers that can have some active subscription. Following the Liveness property we defined in Section 3.1.2, the set of subscribers to be reached by a notification x must comprise all those subscribers whose subscription matches x and is stable at publication time. Then, the diffusion delay is the *maximum* time interval required to diffuse a notification from a broker serving the source publisher to all the interested subscribers in the set. The g function should represent the variation over the time of the number of interested NS participants that received the notification. Again, the maximum diffusion time is obtained when a notification is flooded through all the participants to the system.

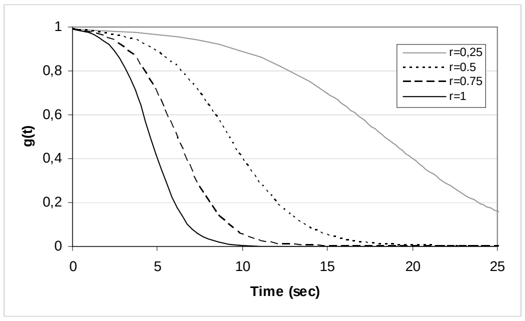
Considering naive flooding as a routing algorithm for subscriptions and notifications allow to be as general as possible in the derivation of the two functions. In other words, the described model represents the worst-case scenario of a wide-range of NS implementation solutions. In particular, all the architectural solutions described in Chapter 2, Section 2.3 roughly follow the above behavior in their worst-case scenarios.

Expressing f(t) and g(t)

The probability that a broker receives a subscription/notification $f(t)$ and $g(t)$ can be represented as the number of brokers in a NS that received a subscription/notification at time t. The message diffusion through a flooding algorithm follows a epidemic-like behavior, that is each participant transmit a subscription/notification only to a subset of other participants, that in turns provide to forward it to other participants. Thus, the following analysis is inspired by the mathematical theory of epidemics [33], often used in the con-



(a)



(b)

Figure 3.6: Analytical trends of f(t) and g(t)

text of gossip-based multicast algorithm to predict the average percentage of participants that will be reached by a multicast message [13].

Let $f(t)$ be the number of NS participants that have received a subscription at time t and N the overall number of brokers in the system. The variation in time of this function is proportional to the number of brokers that does not have received the subscription yet:

$$\frac{\mathrm{d}f(t)}{\mathrm{d}t} = \alpha(t)\,(N - f(t))$$

The value of $\alpha(t)$ is at the same time proportional to $f(t)$, because the more the participants in the system that are aware of the subscription, the faster it spreads. Hence:

$$\frac{\mathrm{d}f(t)}{\mathrm{d}t} = s\,\frac{f(t)}{N}\,(N - f(t))$$

We refer to the constant s as the *subscription update rate*, because it represent the speed of diffusion of an update. s depends on parameters of the NS, such as the time required for a single update, the number of participants updated by a single participant, the time required to process a subscription within each participants.

Solving the above differential equation leads to the following solution:

$$f(t) = \frac{1}{1 + c\, e^{-st}}$$

where c is an arbitrary constant. Since at $t = 0$ only one broker is aware of the subscription (the one that sends it), we can impose $f(0) = 1$ and subsequently obtain $c = N - 1$. The final expression for $f(t)$ is then:

$$f(t) = \frac{1}{1 + (N-1)\, e^{-st}} \tag{3.3}$$

Following a similar procedure, we obtain the following expression for $g(t)$:

$$g(t) = 1 - \frac{1}{1 + (N-1)\, e^{-rt}} \tag{3.4}$$

where r is the *notification update rate*. In general $r \neq s$ because the different algorithms used for routing subscriptions and notifications can result in different diffusion rates. The plots for $f(t)$ and $g(t)$ are shown in Figure 3.6.

s and r are two important parameters that significantly affect the behavior of the system. Unfortunately, an analytical evaluation is not a trivial task. From an experimental measure in a system with 100 nodes configured in a dense topology where all nodes are all physically close to each other, we obtained update rates to be respectively $s = 0.98$ and $r = 1.2$. Details are given in Section 3.3. From the analytical study in Section 3.2.3 it turns out than a system with update rates greater than 1, 5 can be considered a "fast" one, in the sense that with average subscription intervals it can ensure at least the 80% of the notification.

Expressing d

The next step is substituting the above expressions inside equation 3.1, to obtain a general expression for d that depends only from the external operative parameters of the NS.

First we calculate the integrals of $f(t)$:

$$\int_0^{T_{sub}} f(t)\, dt = s\, T_{sub} + \frac{1}{s} \ln \frac{1 + (N-1)e^{-sT_{sub}}}{N}$$

Being $1/(1+(N-1)e^{-sT_{sub}}) \simeq 1$, we obtain

$$\int_0^{T_{sub}} f(t)\, dt = s\, T_{sub} + \frac{1}{s} \ln \frac{1}{N}$$

Analogously:

$$\int_0^{T_{diff}} g(t)\, dt = (1-r)\, T_{diff} - \frac{1}{r} \ln \frac{1}{N}$$

Substituting this expression into equation 3.2.1, results in the following expression for d:

$$d = 1 - \frac{(1-s)T_{sub} + rT_{diff}}{T_{ON}} + \frac{1}{T_{ON}}(\frac{1}{s} - \frac{1}{r}) \ln \frac{1}{N} \qquad (3.5)$$

From the above definitions of T_{sub} and T_{diff} it is clear that it should be $f(T_{sub}) = 1$ and $g(T_{diff}) = 0$. These two values cannot be determined directly from the analytical expressions of $f(t)$ and $g(t)$ because the functions only asymptotically tend to 1 and 0 respectively. However, considering the meaning of the functions, representing the fraction of participants reached by a message, it makes sense to assume only discrete values for them. In particular, the minimal value for the fraction can be $1/N$, then T_{sub} and T_{diff} can be chosen approximately as any values such that $f(T_{sub}) \geq 1 - 1/N$ and $g(T_{diff}) \geq 1/N$. Considering equalities, we obtaining the following approximate expressions:

$$T_{diff} \simeq \frac{1}{r}\ \ln(N-1)^2$$
$$T_{sub} \simeq \frac{1}{s}\ \ln(N-1)^2$$

This allows to rewrite d only in terms of s, r and N:

$$d \simeq 1 - \frac{1}{T_{ON}}\Big(\frac{1}{s} + \frac{1}{r}\Big) \ln N \qquad (3.6)$$

3.2.3 Discussion

In the following we discuss the analytical values obtained by the application of equation 3.6, studying the influence of the various operative parameters over the notification probability. We will consider for simplicity a single parameter α for the update rates (i.e., $\alpha = r = s$). Subsequently, we assume a single stabilization time T_α for both subscriptions and notification.

Figure 3.7 plots the notification probability against the update rate, with different fixed values of T_{ON} and $N = 100$. The sensitivity of the system

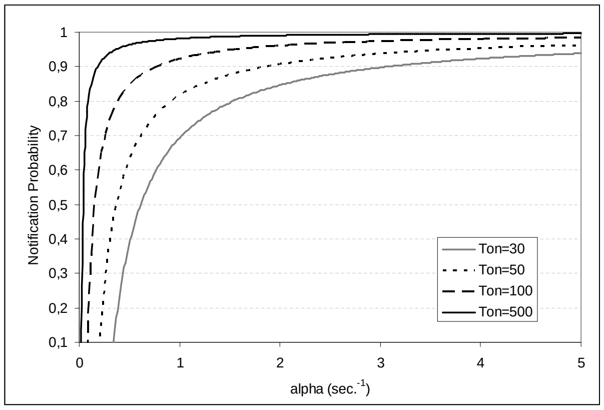


Figure 3.7: Notification Probability vs. Update Rate

to changing values of α strongly depends from the expected value of T_{ON}. With long subscription times ($T_{ON} > 500$) also a slowly responding system ($\alpha < 0,2$) can offer an acceptable level of performance ($d > 0,9$). On the other hand, rapidly changing subscriptions ($T < 50$) require very fast responding system. When $\alpha < 1$, d follows a steep curve. This means that also small variations of α turns out in significant changes to d. However, the plot shows that with values of T_{ON} starting from about 100 seconds, the system reaches relatively high values for d with α higher than about 0,5.

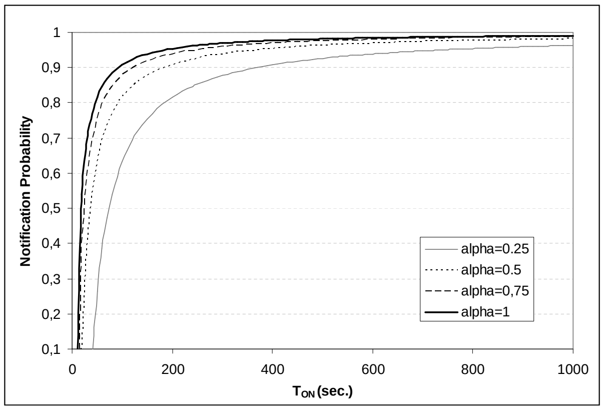


Figure 3.8: Notification Probability vs. Subscription Interval

This results more evidently from the plot in Figure 3.8 showing the notification probability against the variation of T_{ON}, for different values of α and $N = 100$. Moreover, this plot evidences the fact that the more critical behavior, independently from α, occurs when T_{ON} is lower than 60 seconds.

In this case, the system is highly unstable, requiring a careful fixing to achieve acceptable values of d.

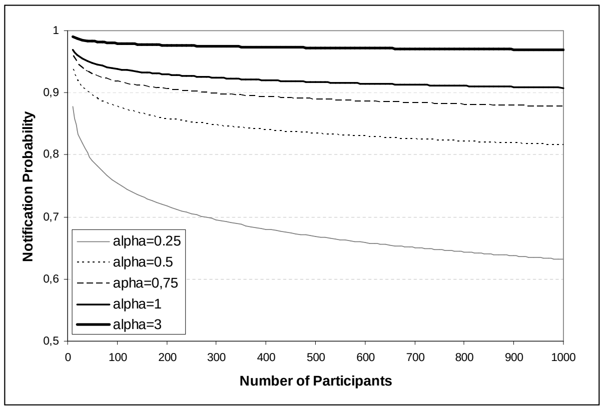


Figure 3.9: Notification Probability vs. Number of Participants

Finally, Figure 3.9 plots the influence of the number of participants N over the notification probability, in correspondence of different fixed values for α (while T_{ON} is fixed to 150). This plot shows that for α higher than 1.5, the system can be considered "fast" because also increasing the number of participants, the notification probability is always higher than 0.9.

In overall, we can say that the analytical model we presented in this section allowed us to make important high-level considerations about the general behavior of a NS. However, a further step is required to fully consider such results trustworthy, i.e. achieve an experimental confirmation. This is the subject of the following Section.

3.3 Simulation Study

In this Section we present the experimental results derived from a simulation of the execution of a real pub/sub system. The objective of the study is to evaluate the actual fraction of the delivered notifications and compare it with the analytical result obtained from the model.

3.3.1 Simulation Details

We carried out our experiments by implementing a prototype of a distributed NS made up of a set of distributed brokers, communicating through point-to-point application-level connections. The system is based on the content-based routing algorithm (CBR) for acyclic peer-to-peer topologies introduced in SIENA [20]. The key idea is to diffuse subscriptions in order to build paths for routing events, so that parts of the network with no interested subscribers are excluded from event diffusion. Simulations were performed by running the NS prototype on top the J-Sim [59] real-time network simulator. We give a more detailed explanation both of the CBR algorithm and of our prototype implementation in the following Chapter.

Network-level topologies are generated using the Georgia Tech ITM topology generator [115]. All topologies follow the Transit-Stub model. The application-level network (i.e. the distribution of the distributed brokers over the network nodes and the links between them) is self-generated by our prototype and follows a random topology that is not influenced by the underlying network topology. This reproduces the typical shift between application-level and network-level that occurs in real deployment of overlay networks. Experiments featured 100 participants (brokers) deployed above networks with 100, 250 and 500 nodes. We considered different deployment scenarios, in order to alter the values of the update rate.

3.3.2 Simulation Results

Update Rates Figures 3.10(b) and 3.10(a) show the results of the experiments over subscription and event diffusion times, with a 100 brokers NS deployed over a 100 nodes network. In particular, Figure 3.10(b) plots the fraction $F(t)$ of brokers at time t that are by a new subscription issued at time 0, and Figure 3.10(a) plots the fraction $G(t)$ of brokers that *have not* received a notification issued at time 0. In practice, these curves are the experimental of the $f(t)$ and $g(t)$ functions, as defined in the previous Section. The shape of the curves coincides with those in Figure 3.6, confirming that epidemics models are appropriate to represent information diffusion processes.

Plots were obtained as following (we consider only T_{sub} as the same can be applied to T_{diff}). We let a subscription start from a random broker and be delivered to all other brokers. This is obviously the worst case scenario that can be encountered for routing. In turn, each broker acts as the source for the subscription. The maximum time required for *all* brokers to receive the subscription, considering all the possible source brokers, corresponds to T_{sub}. Experiments were repeated for 5 times, over different network topologies. The same experiments were carried out with networks of 250 and 500 nodes, producing the same trends, though obviously with different values for T_{sub} and T_{diff}.

The coincidence of the analytical and experimental curves, allows us to evaluate the update rates for subscriptions and notifications. These are the values of s (resp. r) that substituted in equation 3.3 (resp. 3.4) lead to a curve that has the same integral of the one obtained experimentally. The results are reported in Table 3.1.

Network Nodes	s	r	T_{sub}	T_{diff}
100	0.97	1.21	12.20	9.95
250	0.62	0.80	19.06	14.03
500	0.47	0.58	25.44	20.44

Table 3.1: Experimental Values for Update Rates

Note the slight difference between the update rates for subscriptions and notifications. In other words, notification routing turns out to be faster than subscription routing though they both run on the same topology and travel all the brokers. This is due to the fact that at each hop, subscription routing algorithm has to update the local routing data structures, taking more time than the matching operation performed by the notification routing to determine the next hop. Considering that along its execution, the NS is supposed to manage thousands of subscriptions, one can expect the update rate to be subject to changes over the time.

Notification Probability Figure 3.11 shows the number of notifications actually achieved for notifications published at a broker while another broker in the system has an active, matching subscription. These results were obtained as follows: we generated a subscription on a randomly chosen broker, and after a time T_{ON}, a corresponding unsubscription. During the subscription interval, a notification is produced in another random broker. The exact publication time is randomly chosen and follows a uniform distribution inside the subscription interval. We repeated this process over 250 different random subscriber-publisher couples and executed five runs of the experiment each on a different network topology. The whole process was repeated for different values of T_{ON} and over 100 and 250 network nodes, obtaining the curves depicted in Figure 3.11.

The number of notification losses surprisingly does not tend to 0, also for high values of T_{ON}. This demonstrates that the notification loss phenomenon is an important issue in pub/sub systems.

For a final validation of our analytical model, we have to compare the curve obtained experimentally, with the one resulting from equation 3.2.1, by substituting the values for s and r obtained from previous experiment. The percentage error among the experimental curve and the analytical one is shown in figure 3.12. Negative values of the error mean that the analytical value is higher than the experimental one. The plot shows that, apart for lower values of T_{ON} for which the system unpredictability results in a unstable error values[6],

[6]However, we plan to carry out more repetitions of these tests before the final version of the thesis that should stabilize the final results.

the error is bounded in a small 6%. This confirms the overall reliability of the analytical evaluations, apart for smallest values of T_{ON} that are more sensitive to small variations in the results.

Thus, we have shown that our analytical model is able to predict the actual behavior of a NS, expect for a small percentage error. The application of the model with values of s and r that are derived experimentally gives a general estimation of the probability of notification that faithfully reproduces the experimental results.

3.4 Related Work

At the best of our knowledge, the only other research contribution about a formal definition of the semantics of a pub/sub system has been given by Mühl in his PhD Thesis [76]. The thesis presents a framework for specifying a general pub/sub computation and for proving the formal correctness of an important class of diffusion algorithms (*content-based routing algorithms*). Pub/sub computation is abstracted by a sequence of global states interleaved by operations. The characterization of the correct computations of the system is given with respect to the global state. At this level of abstraction, the effect of the delay in the execution of operations is not represented: safety and liveness properties are defined as if any subscription update and notification takes effect instantly.

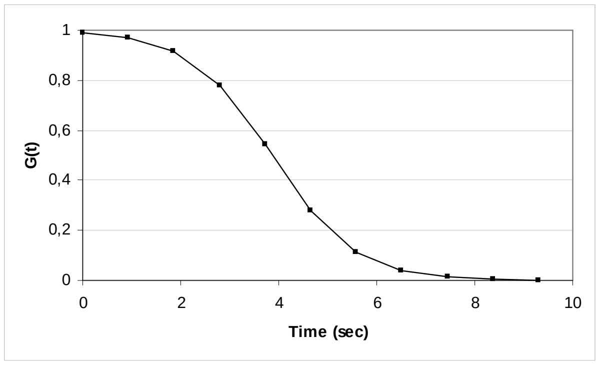
When defining the framework for content-based routing, the author shows that the assumptions may never be satisfied when more update processes occur concurrently. The liveness property is then shown to be guaranteed only for those subscriptions that have been propagated in the whole system (i.e., that are stable, in our terminology).

Though the conclusions are obviously similar, our approach presents several differences with respect to [76]. First of all, the main objective of the work is different: we propose a model to analytically characterize the "quality" of a generic pub/sub system, while Mühl proposes a framework that formally shows the correctness of a class of diffusion algorithms. Besides, our model presents two characterizing aspects, i.e. i) explicitly introducing the pub/sub system in the model and ii) considering the notion of time in the specifications of the system. These two aspects allow to point out the non-deterministic behavior of pub/sub systems at specification level, i.e. independently from any implementation specific such as the deployment type or the diffusion algorithm. On the other hand, we do not give any formal proof but provide only a probabilistic analysis.

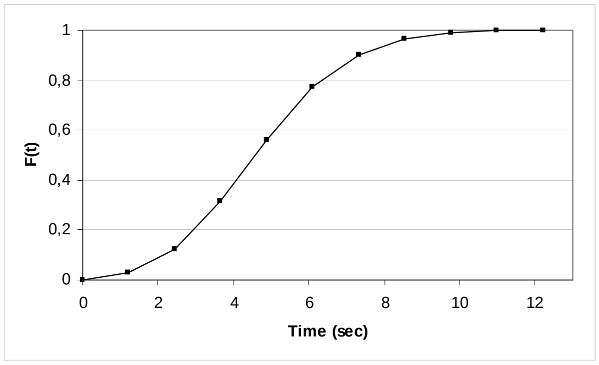
From what concerns the mathematical analysis itself, it was inspired from the mathematical theory of epidemics [33], recently applied to model information diffusion phenomenon like probabilistic reliable multicast [13, 60, 38] or mobile ad-hoc networks [61].

3.5 Concluding Remarks

The models presented in this Chapter allowed us to find out interesting insights about the behavior of a pub/sub system. The computational model highlighted that though the decoupling is usually, and correctly, identified as one of the strongest features of the pub/sub paradigm, it comes at the price of a non-deterministic behavior that results in possible losses of notifications to subscribers. With the following analytical model we gave a method to estimate the cost of non-determinism, i.e. the probability that these losses occur, given a coarse-grained description of the system, in terms of its size (N) and its responsiveness (α). In the future work we plan to investigate how to obtain a more precise characterization for α, that allows a quick evaluation without having to perform measurements.



(a)



(b)

Figure 3.10: Experimental values of T_{diff} and T_{sub}, 100 participants over a 100 nodes network

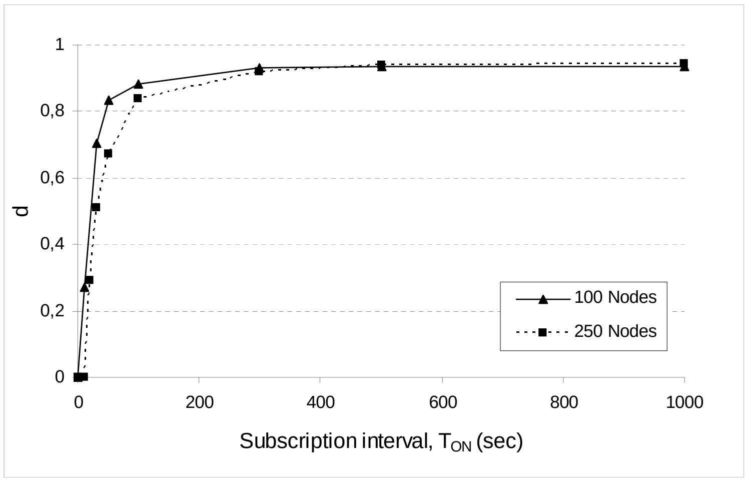


Figure 3.11: Experimental values of d

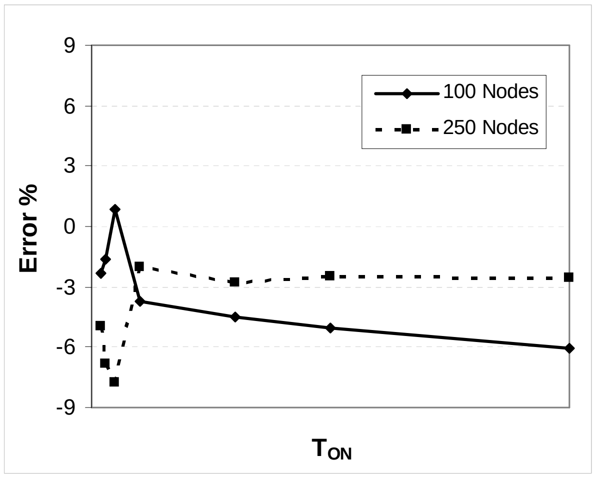


Figure 3.12: Comparing Simulation and Analysis: Evaluation Error

Chapter 4

Self-Organizing Content-Based Publish/Subscribe

Content-based pub/sub is probably the main source of research problems in the pub/sub area. In chapter 2 we pointed out that even though content-based pub/sub is more flexible than its topic-based counterpart, it is notably more difficult to implement: for example the set of intended receivers for a notification cannot be determined a priori but must be computed on a per-notification basis. This makes content-based pub/sub not immediately scalable to systems with a high number of subscribers, publishers and high notification publication rate. Scalable solutions based on an application-level network of brokers, typically exchange messages over a topology formed by *static* TCP-like connections, where static means that once they are created they are not altered during the system execution. However, the fact that application-level connections can be easily created and destroyed, performing a rearrangement of the topology of the network could be exploited to provide the pub/sub system with a self-organization capability, in order to fit to the changing conditions of the applications that run over it.

In this Chapter we consider the classical content-based routing algorithm (CBR), used in SIENA [20], JEDI [29] and REBECA [108] and introduce extensions to it to provide a self-organization algorithms that rearranges the topology of the application-level network to dynamically adapt it to the distribution of subscriptions over the brokers. We introduce a metric, namely *associativity*, that measures the degree of similarity among subscriptions hosted at brokers that are neighbors in the application-level network. The idea is to put brokers sharing similar interest “close” to each other at the application level, in order to further reduce the average number of application-level hops per notification diffusion with respect to the improvement obtained with the "static" CBR algorithm. At the same time, we study the impact of self-organization over underlying topology metrics, since a reduced number of application-level hops does not necessarily results in an improvement of lower-level metrics such as latency, bandwidth or IP hops.

The chapter is structured as follows. Section 4.1 presents the background on content-based publish/subscribe and the motivation of our work. Section 4.2 describes the self-organization algorithm, discussing the concept of subscription similarity and presenting the cost metrics and key mechanisms. Section 4.3 describes the details of the self-organization algorithm. Section 4.4 addresses network-level issues. Section 4.5 presents experimental results. Section 4.6 compares our work with the related works in this research area.

4.1 Background

This section defines the key concepts related to the model of content-based publish/subscribe notification service we refer to in the rest of the paper, including a brief presentation of the content-based routing algorithm on which our self-organization algorithm is based.

4.1.1 Publish/Subscribe Model

We consider a distributed Notification Service based on an application-level network of brokers $\{B_1, .., B_N\}$. Brokers are interconnected through transport-level connections which form an acyclic topology. We consider the practical case of the TCP transport protocol. Then, each broker maintains a set of open TCP connections (*links*) with other brokers (its neighbors). The link connecting brokers B_i and B_j is indicated with $l_{i,j}$ (or $l_{j,i}$) and the set of neighbors of B_i is indicated with $\mathcal{N}_i$. A broker can be contacted by clients, that act as either subscribers or publishers. Each broker stores locally the subscriptions of its subscribers (ADA policy) and communicate with other brokers in order to propagate notifications fired by its publishers.

Subscriptions and notifications are defined over a predetermined schema, constituted by a set of n *attributes*, each defined through a name and a type, where the type can be a common basic type such as integer, real or string. The schema can be graphically represented as an n-dimensional space. In the remainder of the Chapter we consider a structured and fixed space, that is all notifications and subscriptions adhere to a same structure defined by the space itself. We follow this approach for ease of presentation, though the majority of actual systems use an unstructured space. However, all the concepts here can be simply generalized to the unstructured case.

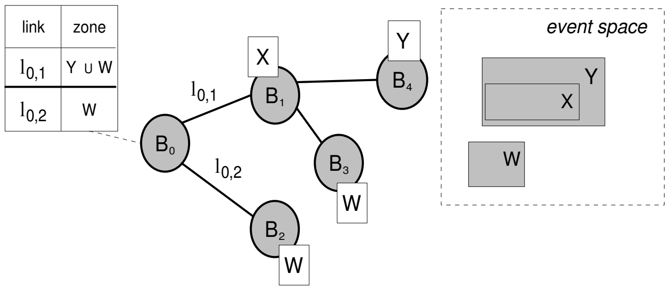


Figure 4.1: Routing Table Example

A notification is a set of n values, one for each attribute, whose type is consistent with that attribute's type. If all values are defined for every attribute, a notification can be considered as a point in the n-dimensional space. Therefore in the sequel we abstract the notion of subscription as a subset of notification inside the space, defined by using a set of *constraints* over the attributes. Constraints depend on the type of attribute, but in general can be represented as range operators. We say that a notification *matches* a subscription if the point it represents falls inside the geometrical *zone* identified by the subscription. Given the range constraints that can be included in subscriptions, a zone is shaped as an n-dimensional hyper-rectangle in the space, aligned along the axis[1]. We define as the *zone of interest* of a broker B_i, denoted Z_i, the union of all local subscriptions at B_i.

4.1.2 Content-based Routing Protocol

As a starting point for our study we consider again the content-based routing algorithms (CBR) introduced in SIENA [20], already described in brief in the previous Chapters. In the following we provide a detailed description of CBR.

Each broker maintains a data structure, namely the *routing table*, that represents a local view of the global subscription distribution. The routing table at a broker B_i has an entry for each neighbor broker of B_i. The generic entry associated to the neighbor B_j is composed by all the subscriptions that are hosted at any broker that can be reached from B_i through the link $l_{i,j}$. In other words, an entry of the routing tables is the union of all the zones of interests of brokers lying "behind" the corresponding link.

Figure 4.1 shows an example of routing table for broker B_0 where outgoing links are denoted $l_{0,1}$ and $l_{0,2}$. The second column of each entry $l_{0,i}$ represents the union of all the zone of interests of brokers which are connected to B_0 through $l_{0,i}$. We denote such union as the zone *covered* by $l_{0,i}$. For example

[1]For simplicity, as far as figures are concerned, we only consider a 2-dimensional space.

$Y \cup W$ is the zone covered by $l_{0,1}$ and W is the zone covered by $l_{0,2}$.

The routing table entries are kept updated by a subscription diffusion process triggered at each subscription change. The CBR algorithm avoids diffusing a subscription by exploiting containment relationships. In particular, when a new subscription S arrives at a broker B_i, if S is not a subset of Z_i, B_i propagates S to the other brokers and adds S to Z_i. When a broker B_j becomes aware of S through a message received from one of its incoming TCP links, namely l, it updates its routing table by adding S to the l's entry in the routing table.

The CBR algorithm for notification diffusion works as follows: each time a notification e is received by a broker B_i either from its local publishers or from a link, it matches e against all local subscriptions and then forwards e *only* through links which can lead to potential e's subscribers. In this case we say that B_i acts as a *forwarder* for e. Forwarding links are determined from the routing table, by performing a matching of the notification against all the entries contained in it.

We denote as *pure forwarder* for e a broker which has no local subscription for e. A pure forwarder introduces one useless TCP hop on the notification diffusion path and requires two useless matching operations, one for the local subscriptions and one for the forwarding step.

4.1.3 Scalability of Content-Based Routing

The advantages of the CBR solution with respect to a naive flooding of each notification over the entire network are obvious, and are showed by several simulation studies [20, 76]. In particular, the main cost metric to be considered is the number of TCP hops travelled by message (see Section 4.2.1 for details).

However, the degree of the improvement of the CBR algorithm over flooding is not constant but strongly depends on the behavior of the specific application. More specifically, the filtering capabilities of CBR perform at their best when the subscription distribution presents some degree of locality. "Locality" intuitively means that subscriptions stored in a same part of the network, are in overall similar each other.

Figure 4.2 depicts the results of a simulation study that shows the effect of subscription locality on the performance of the CBR algorithm. Details on the simulator are given in Section 4.5. The degree of locality varies as follows: the network of brokers is divided in 4 groups of brokers. Brokers in a same group are close to each other. Subscriptions and notifications are defined over a 2D space, divided in four zones. Each zone of the space is assigned to a broker group. Subscriptions are generated through a Zipf distribution centered around 4 different points, each contained in a different zone of the space. Increasing the percentage of locality means having a lower probability

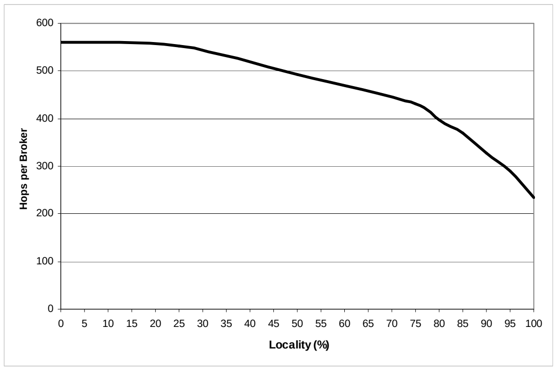


Figure 4.2: Influence of Subscription Locality over CBR Performance

that a subscription is generated at a broker of a different zone. With 100% locality, all brokers in a group host subscriptions of a same zone.

Our study shows that in general increasing the locality implies a strong reduction of the TCP hops per notification diffusion when using a CBR algorithm. This confirms similar results already presented in the literature [76]. On the contrary, when locality is low (i.e., subscribers sharing similar interest are connected to a dispersed subset of distinct brokers which are far each other in terms of TCP hops), the performance gain obtained using the CBR algorithm, with respect to an algorithm that simply floods the network with notifications, becomes negligible.

Current CBR applications rely on the assumption that subscribers in a same part of the network *probably* experience some locality degree. This can be actually true for some applications: for example, in a weather report application, all subscribers in a given geographical area will be probably interested in the report regarding that area. However, the locality assumption does not state in general for any application: for example, in a stock quotes diffusion application, there is no relationship between the physical placement of subscribers and their interest. In this situations, our self-organizing CBR algorithm enforces the locality assumption by exploiting rearrangement capabilities of the application-level network.

4.2 SOCBR: A Self-Organizing CBR Algorithm

In this section, we give the background of the self-organizing content-based routing algorithm (SOCBR). The motivation underlying SOCBR is based on the following question: *how to get a reduction of TCP hops per notification diffusion when the system exhibits low locality?*. We propose a self-organizing algorithm executed by brokers which dynamically arrange TCP connections between pair of brokers in order to directly connect through a TCP link brokers hosting "similar" subscriptions. Therefore, the self-organizing algorithm works in synergy with the content-based routing algorithm (i.e., it can read the CBR table whose data triggers the self-organization, and can modify the CBR tables after the self-organization). We point out that the whole SOCBR algorithm is built by relying at each broker only on the local knowledge of the system provided by information contained in the CBR routing table. This allows to preserve the basic principle underlying the scalability properties of CBR.

In this section we present all the basic concepts that are necessary to realize this idea. First we explain what is the cost metric (the number of TCP hops), i.e. we say what "close" broker means, then how to express "similarity" of subscriptions. Finally, we sketch the algorithm.

4.2.1 The Cost Metric: TCP hops

The main target of the SOCBR algorithm is to do its best to avoid the presence of pure forwarders per notification diffusion, achieving as a consequence the reduction of the TPC hops experienced by a notification to reach all its intended destinations. Lowering the number of TCP hops has a positive impact on the scalability of the system: each broker has to process a lower number of messages, both for the matching and for the forwarding operations.

Reducing the TCP hops metric when considering different diffusion mechanisms on the same topology (i.e., CBR vs. flooding) provides an obvious improvement on the overall network traffic. However, it is important to remark that reducing the TCP hops per notification diffusion could not necessarily lead to a same improvement on network level metrics such as IP hops, latency and bandwidth, when *the route followed by a notification changes.* In other words, rearranging the topology of the network to obtain a new one that is more efficient in terms of TCP hops could not result in a higher efficiency also, for example, for IP hops. In this sense the metric based on the reduction of TCP hops only tries to accommodate the following simple principle: the number of IP hops experienced by a notification diffused over a path formed by many TCP hops is expected to be higher than the ones experienced over paths using less TCP hops. In Section 4.4, we describe an extension of the SOCBR algorithm that addresses also underlying network metrics. Until that point, we describe a network-oblivious version of the SOCBR algorithm, only accounting for the TCP hops metric.

4.2.2 Measuring Subscription Similarity: Associativity

Defining subscription similarity in a content-based system can be very difficult due to the very generic language used for subscriptions. For simplicity, we exploit the geometric interpretation, introduced in Section 4.1, to give a simple yet effective way to solve this problem. In this context, similarity between two subscriptions basically considers the geometrical intersection of the respective hyper-rectangles. However, there could be cases where the difference between two zones can be much higher than their intersection. If the zones are zone of interest of two brokers, the larger the difference between the two zones, the more the notifications that will be received by one broker but not by the other one. Therefore, we define a metric, namely *associativity*, which takes the intersection among zones into account by normalizing this intersection against the overall size of one of the zones. More specifically, let Z and Z' be two zones, we define the *associativity of Z with respect to Z'*, denoted $AS_Z(Z')$ as follows:

$$AS_Z(Z') \doteq \frac{|Z \bigcap Z'|}{|Z|}$$

This notion of associativity can be applied to a broker's zone of interest as well as to the zone covered by a link. In particular, let B_i and B_j be two brokers whose zones of interest are Z_i and Z_j respectively, then the associativity of B_i with respect to B_j is defined as $AS_i(B_j) = AS_{Z_i}(Z_j)$. We also define the associativity of a broker B_i with its neighbors, denoted $AS(B_i)$, as follows:

$$AS(B_i) \doteq \sum_{B_j \in \mathcal{N}_i} AS_i(B_j)$$

We define the associativity of the pub/sub system as

$$AS \doteq \frac{\sum_{B_i \in \{B_1,\ldots,B_N\}} AS(B_i)}{N}$$

Rather than directly using a metric for measuring similarity, an alternative solution would be to create clusters of similar subscribers, similarly to the approaches proposed in [91, 92, 54] for creating multicast group for content-based networking. All current clustering policies proposed in those papers do not provide dynamic adaptivity, that is the clustering policy must be known a priori by all brokers and cannot change during system execution.

Moreover, a clustering policy necessarily introduces an error [78], and there could be notifications that match a group though not exactly matching all the subscribers in the group. In other words, the associativity metric gives a "continuous" measurement of the similarity of brokers' interest, versus the "discrete" one that a clustering policy offer. Since, to the best of our knowledge, our algorithm is the first one in the pub/sub area trying to address the influence of subscription similarity over content-based routing, we chose a precise similarity measurement to isolate the effects of the algorithm itself from the clustering policy.

This lead us to the other important point related to associativity: how to compute efficiently associativity in practice. For the prototype implementation of our algorithm, we implemented algorithms based on computational geometry that given the zones represented by two distinct sets of subscriptions, returns the precise value of associativity. Anyway, the complexity of those algorithms introduces an additional overhead, that can be relevant, especially when the number of subscriptions is high. As part of the future work we plan to study how the use of a less accurate, but more efficient, measure of similarity influences the overall effects of the algorithm.

4.2.3 Algorithm Overview

The SOCBR algorithm follows this simple heuristic: each broker B tries to rearrange the network in order to obtain an increment of its associativity $AS(B)$ while not decreasing the overall associativity of the system AS. Practically this is realized as follows: a self-organization is triggered by a broker B when it detects (through the reading of the CBR tables) that there could be some other broker B', not directly connected to B through a TCP link, that could increase its associativity. The objectives of the self-organization are: (i) to connect B to B' and (ii) to tear down a link in the path between B and B' in order to keep the topology of the network acyclic (a requirement of the CBR algorithm). While point (i) leads to an increment of $AS(B)$, point (ii) can at the same time potentially lead to a decrease of AS. Therefore the algorithm does its best to select the link in the path between B and B' that has the lowest associativity between the two brokers it connects. Self-organization takes place only if it leads to an increase of AS. Increasing the associativity of the whole pub/sub system, intuitively means increasing the probability that two brokers with common interests get close to each other.

4.3 Algorithm Specification

In this section we present in details the SOCBR algorithm. It can be split in four phases: triggering, tear-up link discovery, tear-down link selection, and reconfiguration which includes the CBR routing tables update. During the description of the algorithm we will refer to a running example over the network of brokers depicted in Figures 4.3 and 4.5. The right side of Figure 4.3 shows the zones of interest of each broker, while the cloud indicates a part of the network that does not participate to the self-organization in the example. Before describing the phases in detail, let us introduce some basic notions that will be used in those descriptions.

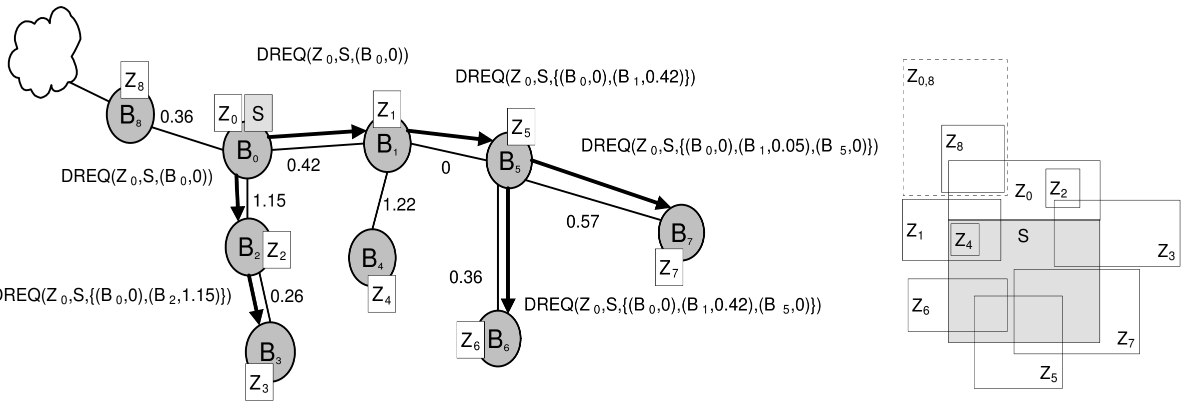

DREQ(Z 0,S,(B 0,0))
DREQ(Z 0,S,{(B 0,0),(B 1,0.42)})
DREQ(Z 0,S,{(B 0,0),(B 1,0.05),(B 5,0)})
DREQ(Z 0,S,(B 0,0))
DREQ(Z 0,S,{(B 0,0),(B 2,1.15)})
DREQ(Z 0,S,{(B 0,0),(B 1,0.42),(B 5,0)})
Z8
B8
0.36
Z0
S
B0
0.42
1.15
Z1
B1
0
1.22
Z5
B5
0.57
B7
Z7
B2
Z2
0.26
B3
Z3
B4
Z4
0.36
Z6
B6
Z0,8
Z8
Z0
Z2
Z1
Z4
S
Z3
Z6
Z7
Z5

Figure 4.3: Tear-up Link Discovery: DREQ Messages.

4.3.1 Basic Notions

Maximum number of TCP links. Each broker can open a bounded number F of TCP links at the same time (F is called the fan-out), and we denote as al_B (with $0 \leq al_B < F$) the number of links available at broker B. For simplicity we assume F equal for all the brokers.

Link weight. Each link $l_{i,j}$ between two brokers B_i and B_j is associated with a weight w that reflects the associativity between the brokers:

$$w(l_{i,j}) = w_{i,j} = AS_i(B_j) + AS_j(B_i)$$

The weight is used to measure the associativity that two brokers gain when a link is created between them and lose when the link is teared down. Figure 4.3 shows the weight of each link[2]. The weight of a link is a measure of the number of notifications that will pass through that link which are of interest to both brokers connected by it.

Hop sequence. A hop sequence is an ordered set of pairs (broker_id, weight) denoted as $HS(B_0, B_\ell)$. More specifically, $HS(B_0, B_\ell) \equiv \{(B_0, 0), (B_1, w_{0,1}), \ldots, (B_i, w_{i-1,i}), \ldots (B_\ell, w_{\ell-1,$ represents the path between B_0 and B_ℓ with the associated weights. Note that the weight associated to B_0 is 0 as it is the first broker on the path. In the pseudo-code we consider an array-like index-based syntax for accessing elements of HS.

[2]This was obtained by applying the algorithm for the computation of associativity to the set of zones depicted in Figure 4.3

4.3.2 Triggering

The algorithm is triggered by a broker B_i when it detects the possibility of increasing its associativity. Specifically, let Z_i be the zone of broker B_i, before the arrival of a new subscription S, and $Z_i^* = Z_i \cup S$ be B_i's new zone of interest. The algorithm is triggered if the following predicate, namely the *Activation Predicate*, is verified:

$$\mathbf{AP} : Z_i^* \supset Z_i \wedge \exists l_{i,j} : AS_{Z_i^*}(Z_{l_{i,j}}) > AS_{Z_i}(Z_{l_{i,j}})$$

For each $l_{i,j}$ satisfying this predicate, a tear-up link discovery procedure is invoked along that link, as B_i suspects that behind it there could be a broker which can increase its associativity.

The triggering phase starts k (with $k \geq 1$) independent discovery procedures, where k is the number of links for which AP is satisfied. Each of these procedures returns the broker B_ℓ with the highest associativity, with respect to B_i, that is located behind the link. Note that the value of $Z_{l_{i,j}}$ can be easily read by looking-up B_i's CBR routing table.

Considering the example in Figure 4.3, let us assume that broker B_0, whose zone of interest is Z_0, receives a new subscription S. The self-organization is triggered, by starting two independent discovery procedure, corresponding to links $l_{0,1}$ and $l_{0,2}$. Link $l_{0,8}$ does not trigger the self-organization, as the associativity of Z_0^* with respect to the zone it covers ($Z_{0,8}$) is not higher than the associativity of Z_0 with respect to $Z_{0,8}$.

4.3.3 Tear-Up Link Discovery

A single discovery procedure works as follows. A request message *DREQ* is sent along the link $l_{i,j}$. The message piggybacks the following information: Z_i, the new subscription S, and the hop sequence HS, initialized to $\{(B_i, 0)\}$.

When a broker B_j receives *DREQ* on its link l (Pseudo-code shown in Figure 4.4), it (i) computes $AS_{Z_i^*}(Z_j)$[3], i.e., the associativity between the broker B_i and B_j; (ii) updates HS by adding (B_j, w_l),i.e., $HS(B_i, B_j) = HS \cup (B_j, w_l)$; (iii) computes the following *Forwarding Predicate*

$$\mathbf{FP} \; : \; \exists l_{j,h} \neq l : AS_{Z_i^*}(Z_{l_{j,h}}) > AS_{Z_i^*}(Z_j)$$

The predicate indicates whenever there is the possibility that a higher associativity can be discovered with a broker behind $l_{j,h}$: if no links exist such that FP is satisfied, then a Discovery Reply message, DREP, is sent back to B along the path stored in HS. In this case, B_j can offer the highest associativity to B_i

[3]Recall that $Z_i^* = Z_i \cup S$

```
TEAR-UP LINK DISCOVERY$Broker B_j$
On Receive DREQ$(Z_i, S, HS)$ from $l$
  begin
    $nreq \leftarrow 0$;
    $HS \leftarrow HS \cup \{(B_j, w_l)\}$;
    $FwLinks \leftarrow \emptyset$;
    for each $l_{j,h}$
      if $(AS_{Z_i^*}(Z_{l_{j,h}}) > AS_{Z_i^*}(Z_j))$
        then
          $FwLinks \leftarrow FwLinks \cup l_{j,h}$;
    end for
    if $(FwLinks = \emptyset)$
      then
        $B_r \leftarrow HS.Last.B$;
        send [DREP$(AS_{Z_i^*}(Z_{l_{j,h}}), Z_j, al_j, HS)$] to $B_r$;
      else
        for each $l_{j,h} \in FwLinks$
          send [DREQ$(Z_i, S, HS)$] to $B_h$;
          $nreq \leftarrow nreq + 1$;
        end for
        $Replies \leftarrow \emptyset$;
  end
```

Figure 4.4: Tear-Up Link Discovery: DREQ Message Handler

compared to the ones that could be provided by brokers behind B_j. The DREP message contains the following information: $AS_{Z_i^*}(Z_{B_j}), Z_{B_j}, al_j, HS(B_i, B_j)$.

Referring to Figure 4.3, note that though B_0 has a non-zero associativity with B_4, the DREQ message is not forwarded to that broker because, as it is clear from the figure, $AS_{Z_0^*}(Z_{l_{1,4}}) < AS_{Z_0^*}(Z_1)$, i.e., the forwarding predicate is not satisfied. For each $l_{j,h}$ satisfying FP, B_j forwards the $DREQ$ and then waits for the corresponding reply message $DREP(AS_{Z_i^*}(Z_\ell), Z_\ell, al_\ell, HS(B_i, B_\ell))$.

When B_j receives the reply from each link (pseudo-code shown in Figure 4.6), it computes the maximum among all the values of the associativity carried in the replies and its own associativity $AS_{Z_i^*}(Z_j)$, i.e. the associativity it can provide. Let as' be the maximum and $B_{\ell'}$ be the corresponding broker; B_j then sends a $DREP(as', Z_{\ell'}, al_{\ell'}, HS(B_i, B_{\ell'}))$ toward B_i. In Figure 4.5, broker B_5 after gathering DREP messages from B_6 and B_7, determines that the broker with higher associativity with B among itself, B_6 and B_7 is the latter. Then, it forwards to B_1 the $DREP$ message received by B_7.

4.3.4 Tear-Down Link Selection

The aim of this phase is to select the link that has to be teared down during the reconfiguration phase. The procedure is activated for each $DREP$ message

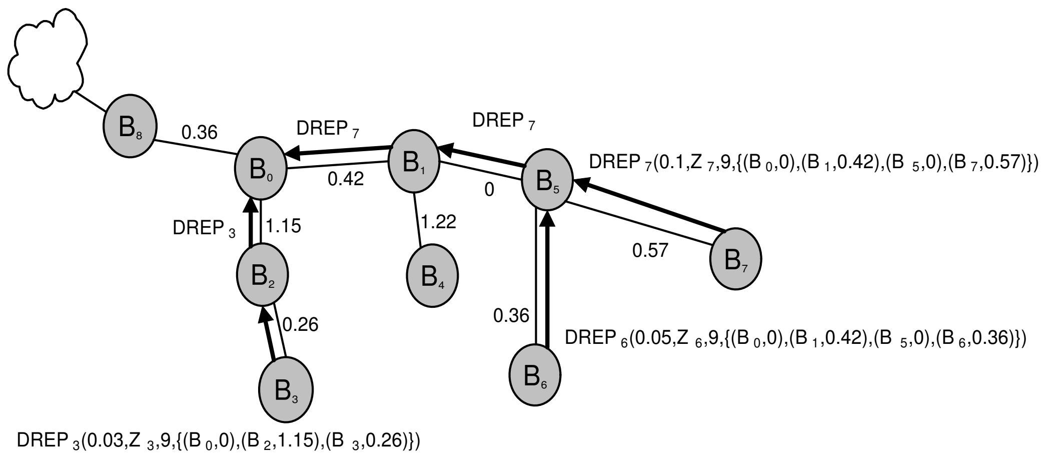

B8
0.36
DREP 7
B0
0.42
B1
DREP 7
0
B5
DREP 7(0.1,Z 7,9,{(B 0,0),(B 1,0.42),(B 5,0),(B 7,0.57)})
DREP 3
1.15
1.22
0.57
B7
B2
B4
0.26
0.36
DREP 6(0.05,Z 6,9,{(B 0,0),(B 1,0.42),(B 5,0),(B 6,0.36)})
B3
B6
DREP 3(0.03,Z 3,9,{(B 0,0),(B 2,1.15),(B 3,0.26)})

Figure 4.5: Tear-up Link Discovery: DREP Messages.

received by the broker B_i that started the self-organization and returns a tear-down candidate link denoted as l_{td}. If no links exist that can be deleted, $l_{td} = NULL$.

A single selection works as follows (pseudo-code in Figure 4.7): let $DREP$ be the reply to the the $DREQ$ message sent along link l. The reply contains the identifier B_ℓ of the broker behind l with the highest similarity with B_i, together with the path stored in HS from B_i to B_ℓ. Let us denote the link to be potentially teared-up between B_i and B_ℓ as l_{tu}.

Clearly, if $al_\ell = 0 \wedge al_i = 0$ the link l_{tu} cannot be created and thus $l_{td} = NULL$.

Otherwise, we have two cases:

1. $al_i > 0 \wedge al_\ell > 0$: In this case both B_i and B_ℓ have available connections and thus they can establish the link l_{tu}, without removing one of their existing links. In this case, l_{td} is the link with the minimum weight in $HS(B_i, B_\ell)$.

2. $al_\ell = 0, al_i > 0$ (resp. $al_i = 0, al_\ell > 0$): in this case l_{td} must be one of B_ℓ's links (resp. B_i), in particular the one that connects B_h to B_ℓ, with B_h being the neighbor of B_ℓ in HS (resp. B_i to B_k, with B_k being the neighbor of B_i in HS), i.e. $l_{td} = l_{h,\ell}$ (resp. $l_{td} = l_{i,k}$); in this way the constraints on the fan-outs are satisfied because the link to be teared down "makes space" for the new link.

The next phase of the algorithm (i.e, the reconfiguration) takes place only if $l_{td} \neq NULL$ and $w(l_{tu}) - w(l_{td}) > 0$. In other words a reconfiguration occurs only if l_{tu} is expected to be crossed by a number of notifications, which interest both brokers directly connected to l_{tu}, greater than the ones which cross l_{td}. This is the heuristic we use in order to ensure that doing the self-organization,

```
TEAR-UP LINK DISCOVERY Broker $B_j$
On Receive DREP($as, Z_\ell, al_\ell, HS(B_i, B_\ell)$) from $l$
  begin
    $nreq \leftarrow nreq - 1$;
    $Replies \leftarrow Replies \cup \{as, B_\ell\}$;
    if ($nreq \neq 0$)
      then
        return ;
      else
        $Replies \leftarrow Replies \cup \{AS_{Z_i^*}(Z_j), B_j\}$;
        $as' \leftarrow Replies.Max(as)$;
        $ThisBrokerIndex \leftarrow HS.IndexOf(B_j)$;
        $PrevBroker \leftarrow HS[ThisBrokerIndex].B$;
        send [DREP($as', Z_\ell, al_i, HS(B_i, B_\ell)$)] to $PrevBroker$;
  end
```

Figure 4.6: Tear-Up Link Discovery: DREP Message Handler, Forwarding Broker

the associativity of the whole pub/sub system AS, does not decrease.

Considering Figure 4.5, the converge-cast phase of DREP messages to B_0 selects B_7 as the candidate to establish a link (i.e., l_{tu}) with B_0, then the algorithm selects $l_{1,5}$ as l_{td}. The reconfiguration will occur as $w(l_{0,7})$ is greater than $w(l_{1,5})$.

4.3.5 Reconfiguration

Let B_i and B_ℓ be the two brokers to be connected by l_{tu} and B_p and B_q be the brokers connected by l_{td} in the path stored in $HS(B_i, B_\ell)$. We have to avoid that during the tear down of the link l_{td}, other links in $HS(B_i, B_\ell)$ are teared down by concurrent executions of the self-organization algorithm. Otherwise, the network of brokers might partition. Therefore there is the need of a locking mechanism to ensure that there is only one tear down operation at a time along the path from B_i to B_ℓ.

This mechanism works as follows: B_i sends a LOCK message along the path towards B_ℓ. A generic broker B in that path executes the following algorithm:

❒ when receiving a LOCK message: if B is involved in other concurrent reconfiguration phases on the same link it received the LOCK message from, it replies with a NACK message. Otherwise we have two cases:

   1. if $B = B_\ell$, B sends an ACK message to the next node towards B_i;
   2. if $B \neq B_\ell$, B sends a LOCK message to the next node towards B_ℓ;

```
TEAR-DOWN LINK SELECTION $Broker\ B_i$
On Receive DREP($as, Z_\ell, al_\ell, HS(B_i, B_\ell)$) from $l$
  begin
    $l_{tu} \leftarrow l_{i,\ell}$;
    if ($al_j = 0 \wedge al_l = 0$)
      then
        return ;
      else if ($al_i = 0 \wedge al_l > 0$)
        $B_k \leftarrow HS[2].B$;
        $l_{td} \leftarrow l_{i,k}$;
      else if ($al_i > 0 \wedge al_l = 0$)
        $LastBrokerIndex \leftarrow HS.IndexOf(B_\ell)$;
        $B_h \leftarrow HS[LastBrokerIndex - 1].B$;
        $l_{td} \leftarrow l_{h,\ell}$;
      else
        $w_{p,q} \leftarrow HS.Min(w)$;
        $l_{td} \leftarrow l_{p,q}$;
        if ($w(l_{tu}) > w(l_t d)$)
          then
            $reconfiguration(B_i, B_l, l_{td}, HS)$;
  end
```

Figure 4.7: Tear-Down Link Selection: DREP Message Handler, Initiating Broker

❒ when receiving an ACK message: B forwards the ACK message to next node towards B_i;

❒ when receiving a NACK message: B forwards the NACK message to next node towards B_i and removes the lock.

Previous algorithm is depicted in Figure 4.8.a when considering the path from B_0 to B_7 of Figure 4.3 and no concurrent reconfigurations on links in that path occur.

Once the path is locked, B_0 sends a CLOSE message to B_q along the path to B_ℓ identifying the link that has to be teared down, namely l_{td} [4]. Figure 4.3.b shows this operation for the path from B_0 to B_7. At this point the link l_{td} is actually teared down.

Finally, the routing tables have to be updated and the locks on the path have to be removed. This operation starts from B_p and B_q and follows the reverse path to B_i and B_ℓ respectively by using UNLOCK messages as shown

[4]This information could also be piggybacked onto ACK messages of previous algorithm. For simplicity, we describe the operation using separate explicit CLOSE messages.

in Figure 4.8.c with respect to the path from B_0 to B_7 [5]. Once the UNLOCK message arrives at B_i, the link l_{tu} can be safely teared up ($l_{0,7}$ in Figure 4.8.c).

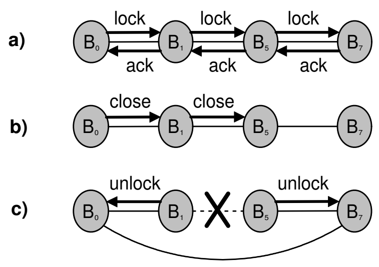


Figure 4.8: Routing Tables Update.

4.4 Addressing Network Proximity

As anticipated in Section 4.2.1, considering TCP hops as a primary metric can provoke some inefficiency over network-level metrics. In this section we concentrate on this problem, presenting a variation of the SOCBR algorithm that addresses this issue.

4.4.1 Network Awareness in Pub/Sub Systems

One of the limitations of current pub/sub systems based on an application-level network is that the topology of the brokers' network does not have any relationship with the topology of the underlying network. This can obviously result in inefficient topologies, like the one depicted in Figure 4.9(a), where the underlying network is indicated with white circles (network nodes) and dashed lines (links), while the brokers network is indicated with grey circles (brokers) and plain lines (TCP links). In this example, a notification travelling from broker B_1 to broker B_3 necessary has to pass through broker B_2. However, the network path from B_1 to B_2 includes the node that hosts broker B_2. Thus, that path will be unnecessary travelled two times.

The problem is then finding a relationship between the application-level proximity, measured as the number of TCP hops, and the network-level proximity, measured with metrics such as the IP hops, latency or bandwidth. For what concerns our SOCBR algorithm, the first version presented in the previous Section follows a network-oblivious approach, trying to achieve only a reduction in TCP hops. In essence, a SOCBR run starts from a random topology and finds out another random one. This in theory should not cause any

[5]Notifications in transit on the path during the reconfiguration, will not be lost as they will "follow" the reversal of the routing tables as this takes place.

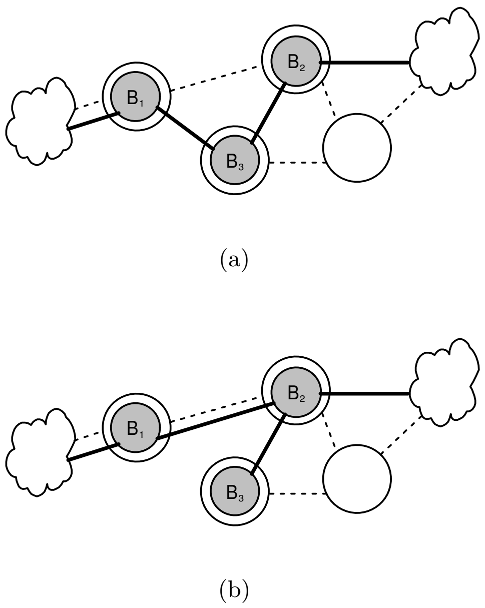


Figure 4.9: Application-Level and Network-Level Topologies

harm to the average network-level performance of the system if brokers in the initial topology do not feature any significant degree of physical proximity among themselves.

Nevertheless, in practice, one must consider that application-level networks in pub/sub systems are typically supposed to be built “by hand” by the application developer. It is then probable that in the initial topology there is some degree of physical proximity among the brokers. This proximity is not taken into account by the SOCBR algorithm and can be spoiled after several runs of it, subsequently resulting in a degradation of network level performance metrics. This is confirmed by our experimental study.

Referring again to Figure 4.9(a), that topology could have been obtained by a self-organization that led broker B_1 to connect with B_3, because it was the one with higher associativity with it. If also B_2 had a high associativity with B_1, many notifications will traverse the application-level path composed by B_1, B_2 and B_3, following the inefficient network path that we discussed before.

Though the focus of our work is improving the application-level performance of a content-based pub/sub system, we have to consider this negative side-effect of self-organization upon network-level metrics. In the remainder of this section we explain how we addressed those issues in a network-aware variation of the SOCBR algorithm.

4.4.2 pbSOCBR: Network-Aware Self-Organization

In the following we describe a variation over the SOCBR algorithm, namely pbSOCBR (proximity-bounded Self-Organizing Content-Based Routing), realizing a heuristic based on a proximity bound over the possible responses. The principle of the pbSOCBR algorithm is the following: reconfiguration can occur only with those brokers whose network-level distance with the source broker is less than a reference value d_b. In other words, a broker B starting the reconfiguration will choose as its new neighbor the most similar one having a network distance from it in the range delimited by d_b. Network distance can be measured indifferently with any metric, either IP hops or latency or bandwidth, depending on the specific requirements of the application.

The simple heuristics realized by the pbSOCBR can avoid anomalies such as the one depicted in Figure 4.9(a). In that example, considering the distance measured with IP hops and a bound d_b of 3 hops, B_1 will not start a reconfiguration with B_3 because it discovers it as "far". It will rather choose B_2 instead, obtaining the topology shown in Figure 4.9(b). This solution trades some unnecessary TCP hops, due to the notifications directed from B_1 to B_3 and not matching B_2's subscriptions, for a general higher network-level efficiency.

The pbSOCBR variation affects only the code of the tear-up discovery phase (pseudo-code shown in Figures 4.10 and 4.11). Each broker involved in the discovery phase having, will probe its distance from the source of the self-organization and verify if it is higher than d_b. In this case it will simply avoid to propose itself as a candidate (Figure 4.10, Lines 14–16, and Figure 4.11, Lines 9–11), letting the discovery continue like in the ordinary case.

Another modification is made to the weight of links, that now includes both mutual associativity and distance between nodes. Being a the associativity between the source node and its new neighbor, the link to be cut in reconfiguration is chosen among the ones that have associativity lower than a as the one with highest distance from the source of the self-organization.

The performance of pbSOCBR obviously depends on the choice of l_b. If it is too high, the algorithm will find brokers that are "far" in the network, possibly spoiling the initial proximity of broker, if any. If it is too low, the algorithm will choose the new neighbor inside a restricted set of brokers, hardly finding one with a good associativity. Then, the subsequent reconfiguration will have a negligible effect on the TCP hops metric.

```
TEAR-UP LINK DISCOVERY Broker $B_j$
On Receive DREQ$(Z_i, S, HS)$ from $l$
  begin
    $nreq \leftarrow 0$;
    $HS \leftarrow HS \cup \{(B_j, w_\ell)\}$;
    $FwLinks \leftarrow \emptyset$;
    for each $l_{j,h}$
      if $(AS_{Z_i^*}(Z_{l_{j,h}}) > AS_{Z_i^*}(Z_j))$
        then
          $FwLinks \leftarrow FwLinks \cup l_{j,h}$;
    end for
    if $(FwLinks = \emptyset)$
      then
        $B_r \leftarrow HS.Last.B$;
        if $(Distance(B_j, B_i) > d_b)$
          then
            $HS \leftarrow HS - \{(B_j, w_\ell)\}$;

        send [DREP$(AS_{Z_i^*}(Z_{l_{j,h}}), Z_j, al_j, HS)$] to $B_r$;
      else
        for each $l_{j,h} \in FwLinks$
          send [DREQ$(Z_i, S, HS)$] to $B_h$;
          $nreq \leftarrow nreq + 1$;

        end for
        $Replies \leftarrow \emptyset$;
  end
```

Figure 4.10: pbSOCBR: DREQ Message Handler

```
TEAR-UP LINK DISCOVERY$Broker\ B_j$
On Receive DREP($as, Z_\ell, al_\ell, HS(B_i, B_\ell)$) from $l$
  begin
    $nreq \leftarrow nreq - 1$;
    $Replies \leftarrow Replies \cup \{as, B_\ell\}$;
    if ($nreq \neq 0$)
      then
        return ;
      else
        if ($Distance(B_j, B_i) < d_b$)
          then
            $Replies \leftarrow Replies \cup \{AS_{Z_i^*}(Z_j), B_j\}$;
        $as' \leftarrow Replies.Max(as)$;
        $ThisBrokerIndex \leftarrow HS.IndexOf(B_j)$;
        $PrevBroker \leftarrow HS[ThisBrokerIndex].B$;
        send [DREP($as', Z_\ell, al_i, HS(B_i, B_\ell)$)] to $PrevBroker$;
  end
```

Figure 4.11: pbSOCBR: DREP Message Handler, Forwarding Broker

4.5 Simulation Study

This section presents the simulation results for the SOCBR algorithm. In the simulation study we observe a prototype pub/sub system subject to a predefined stimuli (e.g., publishing of notifications, issuing of subscriptions, etc.) and compare its performance obtained by a SIENA-like CBR algorithm with and without the self-organization. The values reported in the plots are the mean of 5 independent runs; the confidence interval is not shown as its value is always lower than 2% of the mean. The prototype is integrated within the J-Sim real-time network simulator[59] to simulate the behavior of large networks of brokers.

4.5.1 Implementation Details

Our pub/sub prototype has been implemented in Java and is actually an implementation of SIENA, enhanced with a full implementation of the SOCBR algorithm. Simulations were performed by running the prototype onto the J-Sim simulator. J-Sim is a real-time network simulator based on a component model. Each node of our simulator hosts components that simulates the TCP/IP stack, starting from the network level. The Distance-Vector protocol was used for IP-level routing.

Our pub/sub prototype consists basically in a component implementing a broker that is plugged onto the TCP level, exploiting sockets for creating application-level links among each broker instance. However, the design of the prototype is completely independent from any specific feature of the J-Sim simulator and this will allow us in future work to deploy a stand-alone running version of the same prototype.

The prototype relies on matching and containment algorithms implemented upon a R-Tree data structure [55]. R-Trees provide efficient handling of range-based data, such as the ones we used in our content-based subscription model. Both the routing table and the internal subscription table are represented as two separate R-Trees. The insertion and the removal of subscriptions, as well as notification matching are completely managed by an external R-Tree library.

4.5.2 Simulation Scenarios

The implemented software includes also a scenario generator that creates the network-level topology and the script for randomly generating subscriptions and notifications. For what concerns the network-level topology, the generator can take as input a file obtained by the GT-ITM topology generator, describing a Transit-Stub or Flat network, and convert it to a J-Sim TCL script.

We simulate a network composed of N network nodes, with one broker running on each node. The physical-level and application-level topologies are independent (i.e. two nodes connected at physical-level do not necessarily host two brokers connected at application-level). In general, the physical topology is generated as a flat random graph, except where differently indicated. The application-level topology with $k + 1$ brokers is obtained from a k-brokers topology by starting a new broker process on a node and connecting it to one of the brokers inside the network; this broker is selected randomly.

In the rest of this section, if not differently specified, we consider $N = 100$ nodes, each with a maximum fan-out degree $F = 10$. A broker can have both publishers and subscribers attached to it; however, we do not model explicitly the number of attached clients.

Simulation scenarios are sequences of subscriptions changes and notification publications, over a bi-dimensional space with two numerical attributes. Each subscription is a rectangle characterized by the center, generated from a Zipf distribution, and the extension over the two dimensions, generated from a uniform distribution. This method simulates the aggregation of subscriptions around specific points in the space, that occurs in most real-world applications [110]. Notifications are generated using an uniform distribution over the space.

Publications or subscription changes are injected in randomly chosen brokers. The ratio between the number of publications and the number of subscriptions is denoted as R, i.e. a new subscription change occurs every R publish operations. R is a single traffic parameter enclosing the contributions

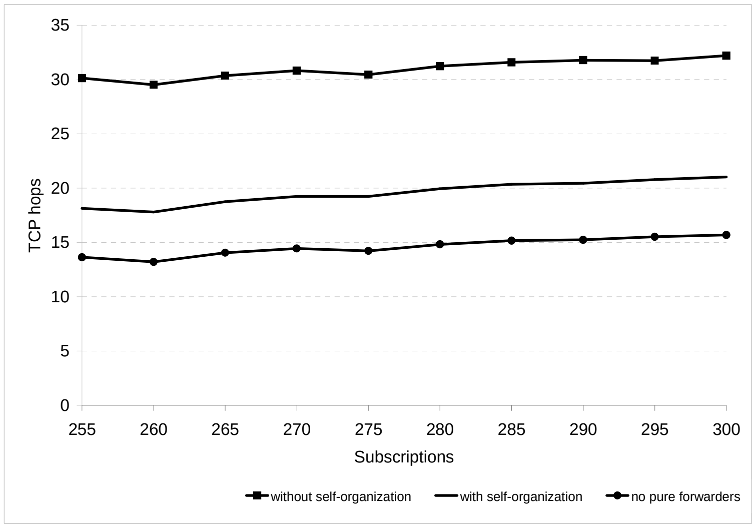


Figure 4.12: Number of TCP hops per notification as a function of subscriptions for $R = 100 : 1$.

due to the two independent traffic sources of pub/sub systems, i.e. publishers and subscribers. We always consider $R > 1$, that is publications are injected more frequently than subscriptions. This is a common, reasonable assumption in pub/sub systems.

4.5.3 Experimental Results

Routing Performance The main effect we expect from self-organization is a reduction in the number of TCP hops required per notification. Figure 4.12 shows the average number of TCP hops per notification obtained with and without the self-organization algorithm. These two values are compared with the ideal value obtained when no pure forwarder brokers are involved in the notification routing: in this case each notification is delivered to each recipient in a single TCP hop. This is obviously an ideal situation that represents the minimum number of brokers involved in routing, but it gives a clear reference value for the improvement obtained by applying the self-organization algorithm: due to the self-organization the number of pure forwarders is reduced by a 70% obtaining an average value that is very close to the minimum. Values for this plot are obtained for $R = 100 : 1$ in a window of 50 subscriptions. Sliding this window does not lead to different results.

The improvement in the TCP hops follows from the enhancement of the associativity metric. Figure 4.13 shows the associativity as a function of the number of subscriptions in the system, with and without the self-organization

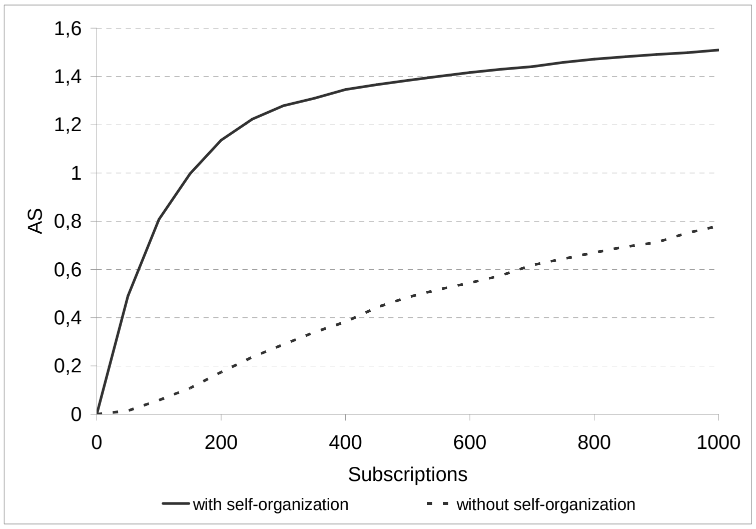


Figure 4.13: *AS* vs. subscriptions.

algorithm. When self-organization is not used, the associativity at a broker B obviously increases while increasing the overall number of subscriptions. However, if self-organization is allowed, a high associativity value is achieved very quickly.

Self-organization Overhead. The improvement in routing performance due to the SOCBR algorithm, comes at the price of additional network traffic introduced by the algorithm itself. Obviously, this overhead can be balanced by the gains illustrated in Figure 4.12 only when publications are injected in the system more frequently than subscriptions. Though this is a common assumption under most pub/sub systems, it is necessary to exactly identify the conditions in which self-organization becomes convenient.

Figure 4.14 reports the average number of messages per operation (either publication or subscription) against the pub/sub ratio R. Such messages include all the messages generated by the system for notification diffusion, subscription routing and self-organization. For a ratio as low as $R = 10 : 1$ the self-organization cost is not outweighed by its benefits. As the ratio increases, the cost decreases accordingly, quickly reaching a reduction in the overall number of messages of about 30% with respect to the case when no self-organization is performed. The break-even point lays at about $R = 12 : 1$. We stress the fact that this ratio is far below the common working conditions assumed for pub/sub systems. The figure shows how the self-organizing algorithm can help the CBR algorithm to scale with respect to increasing load due

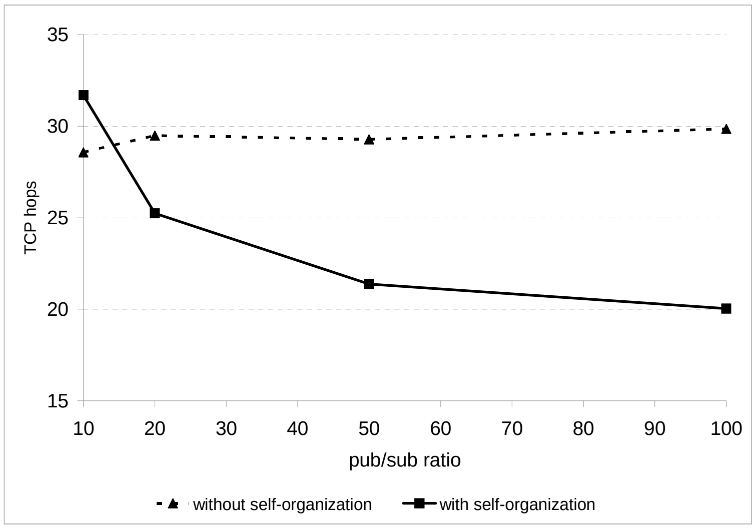


Figure 4.14: Number of messages per operation (publish or subscribe) as a function of the pub/sub ratio R

to publications. In other words, when the publication rate grows, the number of TCP hops per notification decreases.

Figure 4.15 reports the overall average number of self-organization messages required per subscription when more and more subscriptions are injected in the system. The plot shows that the impact of self-organization over a subscription change follows a sub-linear trend, stabilizing (and slightly descending) when only a low number of subscriptions is present in the system. This result points out the scalability of the self-organizing algorithm (with respect to the cost of reconfiguration) while increasing the subscription load.

TTL Strategy. In our experiments we have measured that about 90% of messages of a self-organization are due to the discovery phase (DREQ/DREP messages). To reduce this cost one straightforward way consists in limiting the scope of the discovery, by setting a suitable value for the TTL field in the DREQ messages[6]. Clearly, there is the risk that the broker discovered in this way is not the best possible at that time, or even that no self-organization is triggered at all while there could be such a possibility.

Figure 4.16 shows the average number of messages generated for each operation by varying the TTL value[7]. Note that the plots with TTL=0 and

[6]Please note that the TTL is a bound over the application-level distance. The bound in network-level distance, realized in the pbSOCBR variation, is considered in the last set of experiments in this section.

[7]The average diameter of the network topologies used in the simulations with 100 nodes

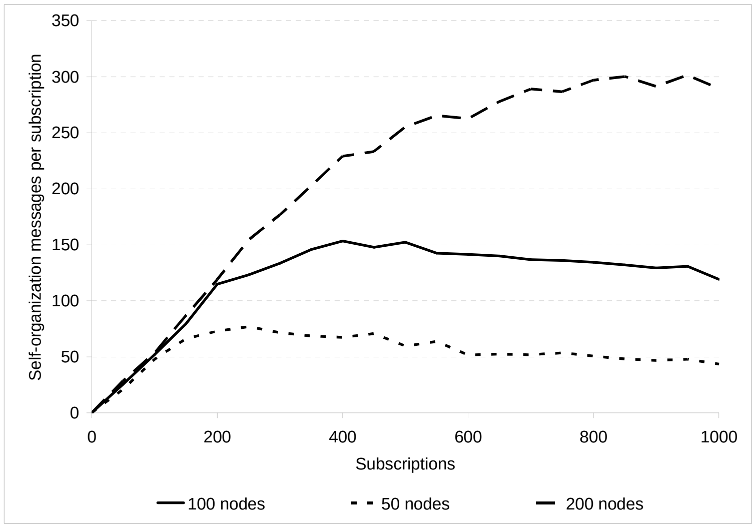


Figure 4.15: Average number of self-organization messages per subscription.

TTL=∞ correspond to the plots of Figure 4.14. Though the limited scope of the discovery phase brings to suboptimal configurations, the subsequent reduction of discovery messages allows to achieve an overall improvement for low values of R. As R increases, this is no longer true: in the suboptimal topology the performance of routing quickly degrades, tending to that experienced in absence of self-organization.

This solution represents an interesting variation, as a broker could estimate locally R and the diameter of the network (in terms of TCP hops) and decide dynamically the best TTL value to associate with each discovery procedure.

Network Latency We studied the effect of the application of self-organization algorithm over network-level proximity of brokers. We chose to use latency as network proximity metric, because it is a measure of the reactiveness of the pub/sub system in delivering notifications, that for what we saw in previous Chapter strongly influences the probability of reaching all the subscribers.

All the experiments were performed over a transit-stub topology, letting CBR and SOCBR run in identical conditions (i.e., same topology, same subscriptions and same notifications). This was repeated for 5 independent runs. Also the initial topology of brokers is the same and it is completely independent from the underlying network topology. This is a very important point: this procedure is different from the common simulation settings for pub/sub systems, that does not consider the difference between application-level and

was about 16.

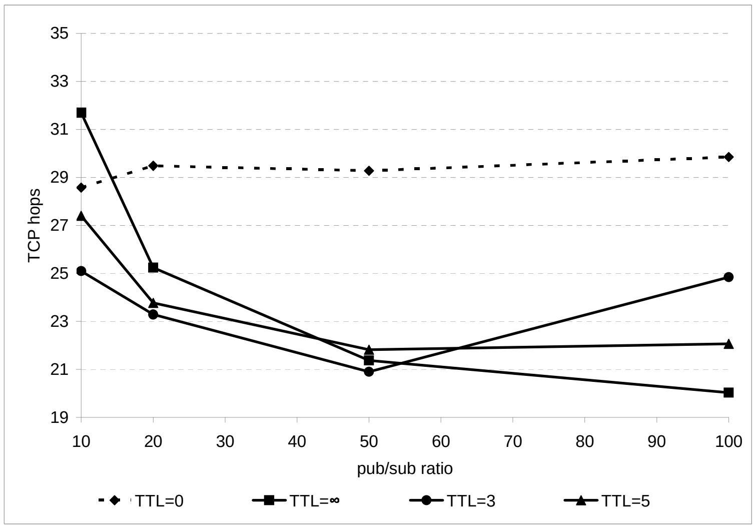


Figure 4.16: Performances of the algorithm with a limited TTL.

network-level topologies. For example, experiments in [21] rely on a GT-ITM topology but the brokers' topology exactly overlaps to the network topology. The results obtained under this assumption are valid whether one has to consider only application-level messages. When measuring network-level performance, like in our case, that simple solution is no more feasible. At the best of our knowledge, this is the first study of the network performance of a pub/sub system based on an application-level network (i.e., without an underlying overlay network infrastructure).

The results obtained from this experiment are shown in Figure 4.17, plotting the average end-to-end latency experienced per each notification, that is the time elapsed from the notification source to the last subscriber. The SOCBR algorithm induces a 1.5-2 seconds overhead over the average latency, equivalent to a 50% increase. Moreover, the curve does not follow any specific trend, confirming the randomness of the effect over the network.

Though this is a very poor result, it probably stems from the fact that in the initial topology created by our prototype it is probable that neighbor brokers may find themselves either in a same subnet or directly connected to core routers. This allowed us to obtain in a simple way an application-level topology in which the side-effect of SOCBR is more evident. In other words, we created a sort of worst-case scenario, and the result is clear: shuffling an application-level topology in which brokers experience some degree of network-level proximity, can result in a serious, unpredictable drawback over the general network-level performance.

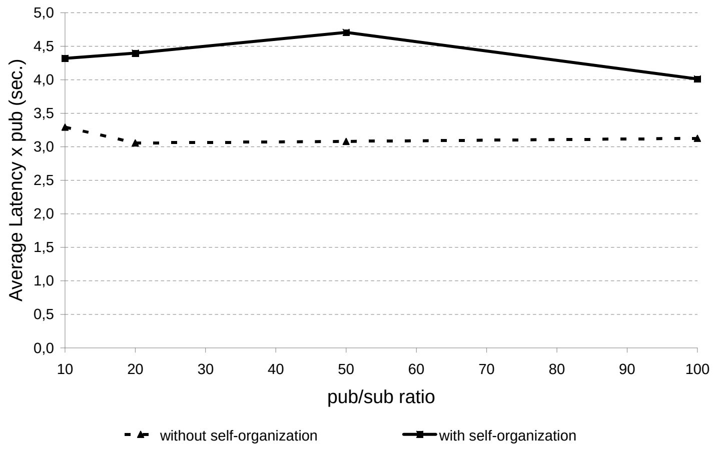


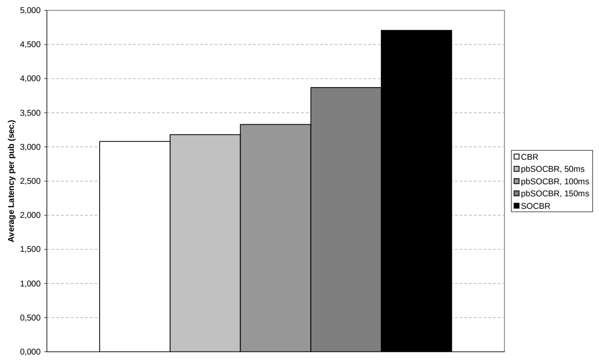
Figure 4.17: Average latency per notification

Latency Bound Figures 4.18(a), 4.18(b) and 4.20 show the results obtained by executing the pbSOCBR variation described in Section 4.4. The main problem in applying pbSOCBR is to estimate the value for d_b that results in the best trade-off between the number of hops saved and the latency overhead induced on the network. Our estimation started from the observation of the maximum end-to-end latency from one node to another in the transit-stub topologies used for the simulations. This was approximately 200ms. A rough estimate for this value in a real-world application can be obtained by measuring the latency between the two brokers that are supposed to be more geographically “far”.

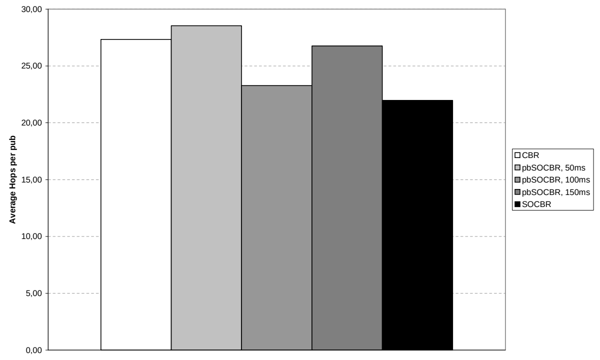
Thus, we let pbSOCBR run with values for d_b of 50ms, 100ms and 150ms, fixing the pub/sub ratio to 50 : 1. The comparison with runs of CBR and SOCBR under the same scenarios (i.e. same subscriptions, notifications and topology) are depicted in the plots in Figures 4.18(a) and 4.18(b). It is evident how by changing d_b the balance between TCP hops and latency is altered. With d_b = 50ms self-organization has a small effect, with a TCP hop value similar to the one obtained with no self-organization. At the same time latency is not affected. On the contrary, with d_b = 150ms the results are very similar to those obtained for regular SOCBR. A good trade-off is obtained with d_b = 100ms: in terms of TCP hops the result is close to the SOCBR performance but the network latency is only about 0,3 seconds more than in CBR.

We focus on this latter result in Figures 4.19 and 4.20 depicting respectively the trend of the number of TCP hops per notification[8] and the average latency

[8]This include the self-organization overhead and also the further pings required for latency measurement.



(a)



(b)

Figure 4.18: Comparison of CBR, SOCBR and pbSOCBR

per notification with increasing values of R. The results are compared to those obtained for CBR. For what concerns the TCP hops metric, at increasing values for R the average hops per notification degrades more gracefully than in SOCBR. The break-even with respect to the pure CBR moves to a value of about $R = 30 : 1$, but for R=100 the gain over CBR is still a 30% less. In terms of latency, the effectiveness of the pbSOCBR heuristics with respect to the network-obliviousness of SOCBR is clear. The overhead is reduced to a maximum of 10%, that in absolute terms is only 0,3 sec., and the trend of the curve is more regular, denoting a generally stable behavior.

In overall, the experimental results show that by limiting the scope of discovery to half the diameter of the network (in terms of latency), pbSOCBR

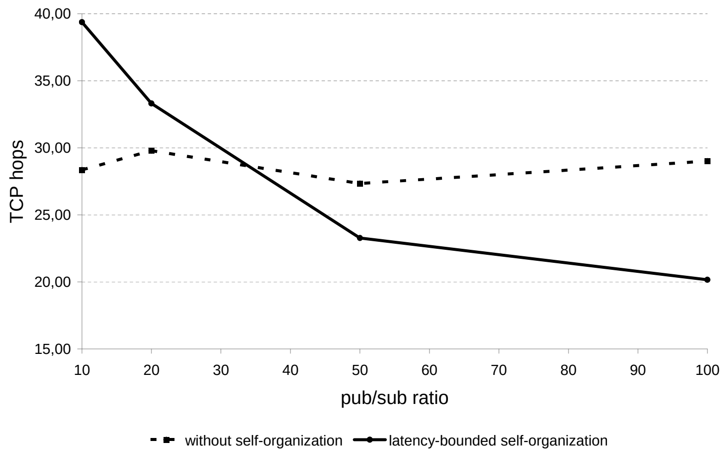


Figure 4.19: Average TCP Hops per notification in pbSOCBR

still achieves the significant improvement on the TCP hop metrics of SOCBR but without affecting much the underlying network metric.

4.6 Related Work

The idea of exploiting self-configuring application-level networks in a pub/sub interaction is starting to become common to several contributions both in the research fields of content-based routing and peer-to-peer overlay networks. Anyway there are many differences between such approaches and the work presented in this paper. We will clarify them in the remainder of this Section.

Reconfiguration in Pub/Sub Systems At the best of our knowledge, the work that more closely resembles our approach to reconfiguration is a paper by Cugola et al. [82], presenting an algorithm for topological reconfiguration in content-based publish/subscribe. The basic difference with our SOCBR algorithm is in the overall motivation for reconfiguration: in [82], authors assume that some link may disappear and others appear elsewhere, because of changes in the underlying connectivity (as it happens for example in mobile ad-hoc networks). "Reconfiguration" in this case means fixing routing tables entries that have become not valid after the change in topology. Differently, in our approach we *induce* a reconfiguration (or, better, a self-organization): that is, it is the algorithm itself that creates and drops links to change the overall topology to a new one that is more favorable for notification routing. Efficient update of routing tables is only a part of our approach. Another difference between the two works is the subscription model used for the simulations:

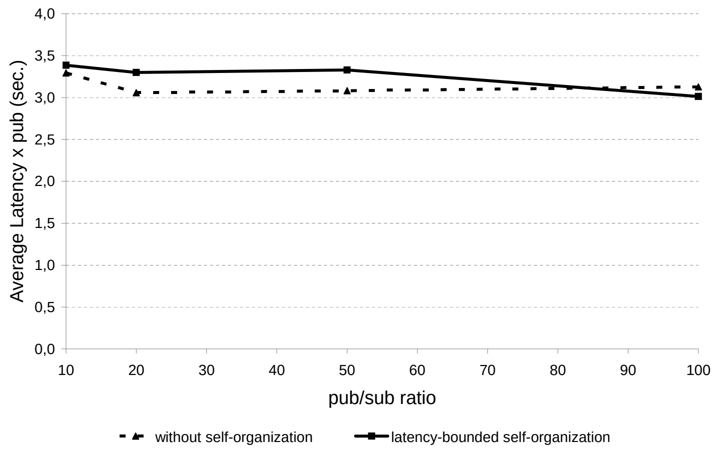


Figure 4.20: Average latency per notification in pbSOCBR

in [82] a simplified content-based model is considered, where subscriptions contain a single char-typed attribute, with only equality predicates. In our work we use a general content-based model, extensible to all data types and supporting also range constraints.

Pub/Sub on Overlay Networks Reconfiguration in application-level networks has been widely studied in the context of structured peer-to-peer overlay network infrastructures, such as Chord [105], Pastry [95] and Tapestry [116]. Such systems offer high-performance point-to-point routing for large scale networks. Their self-reconfiguration capability is exploited for fault-tolerant routing and for adjusting routing paths with respect to metrics of the underlying network.

Overlay network infrastructures represent a general connectivity framework for many classes of peer-to-peer applications, including publish/subscribe. For example, Scribe [22, 94] and Bayeux [117] are two topic-based publish/subscribe systems built on top of two overlay network infrastructures (respectively Pastry and Tapestry). Only recently in ([84] and [108]) the same idea is being applied to two content-based systems.

With respect to the aforementioned contributions, our approach to reconfiguration differentiates again in two senses: i) it is specific for pub/sub applications and ii) it operates *on the same level* as content-based routing. In other words, rather than basing the reconfiguration on metrics of the underlying network, we start from an application-level routing algorithm and add a dynamic behavior that relies on the algorithm's own information (i.e., the routing tables) to improve its performance.

Topology-Aware Overlay Network As far as the problem of building a topology-aware application-level overlay is concerned, several contributions exist in the overlay network literature. Pastry addresses network proximity (as described in [24, 23]) through a heuristic that chooses the neighbors for any node in the network as the ones that are the closest at network-level to the node (according to a generic distance metric). The solution we implemented in the pbSOCBR algorithm presents some resemblance with the Pastry heuristics. An alternative solution is exploited in CAN, presented in [88]: a set of nodes in the network are selected as reference nodes for the distance (*landmark nodes*). Each new node measures the distances with the landmarks, obtaining a vector that is ordered according to the distance. All nodes with the same ordering are supposed to be close within each others and are set as neighbors on the overlay. Other techniques for building topology-aware overlays are described in [62, 113, 106].

In all the papers above the overlay is built by only taking into account the underlying network topology, without considering, as we do in pbSOCBR, also application level information. Though this is consistent with the general purpose nature of overlay network infrastructures, it can result in useless routing hops when multicasting messages.

4.7 Concluding Remarks

The latest research contributions on content-based routing [84, 108] exploit peer-to-peer overlay network as a communication facility to overcome the main problem of the pure application-level approach: the fact that the application-level topology is oblivious of the underlying network topology. Anyway, we believe that routing performance under the pure application-level approach can still be improved by introducing self-organization techniques directly in the application-level network, rather than leveraging the additional overlay layer.

This is the spirit of the self-organization algorithm, SOCBR, and of its network-aware variant, pbSOCBR, that were the subject of this Chapter. The results presented in this thesis are in this sense promising: the SOCBR algorithm cuts off almost all the useless routing steps and it allows to obtain a self-organizing content-based that adapts to the changing distribution of subscriptions. The pbSOCBR goes one step further by finding an accommodation between the requirements from "above" (the distribution of subscriptions) and the ones from "below" (the topology of the network). We see these results only as a first step in what we believe to be a promising research direction (see Chapter 6 for details on planned future work).

Chapter 5

Publish/Subscribe at Work: The DaQuinCIS Project

Pub/sub systems represent a general-purpose solution for information dissemination, that can fit several scenarios which require an asynchronous many-to-many communication. The models and algorithms that we studied in previous chapters apply to a general pub/sub model with non restrictive underlying assumptions. However, when considering specific applicative settings such generic model may not fit in two senses: on one hand, the application context can impose technical constraints which has to be accounted in the architecture of the system; on the other, it may allow to loosen the generality of the model, making it possible to introduce simplifications and optimizations.

In this chapter we study the application of pub/sub to the management of quality of data in information systems composed by a set of autonomous and cooperating organizations (Cooperative Information System - CIS). This work was carried out in the context of the DaQuinCIS project [98], a research project focused on methodologies and software tools for the management of data quality in CIS's. The software contribution of the DaQuinCIS project is an architecture for managing data quality in CIS's. The architecture aims at avoiding dissemination of low qualified data through the CIS, by providing a support for data quality diffusion and improvement. The architecture is supported by a specifically-designed data model for the representation of both data and quality metadata.

In particular, we focus on the design of one component of the DaQuinCIS architecture, namely the Quality Notification Service (QNS)[67]. The QNS is a pub/sub service that can be used to asynchronously notify users for changes in the quality of data within the CIS.

Under the point of view of pub/sub research, the design of the QNS within the CIS scenario poses several challenges:

❒ the need for a novel subscription language, based on the content-based model, but adapted to the data model used inside the whole CIS;

❒ the scalability of the service to a large number of users managing massive amounts of data;

❒ the deployment of the service in a system where multiple organizations must communicate while preserving their independence.

This latter point introduces a strong constraint on the architectural choice. In particular, the DaQuinCIS architecture relies on a service-based integration model, that is, each organization can communicate with other organizations only by invoking the services it exports. Thus, all the components inside the architecture, including the QNS, must be built upon the service-oriented paradigm [79] and it is not possible to directly apply one of the design solutions considered in the previous Chapters. However, the CIS model in some senses is more simple than the generic pub/sub model we considered until this point. This offers the opportunity for optimizing some aspects of the QNS design.

This chapter describes how we addressed all the aforementioned issues in our design of the QNS. After a general overview of the DaQuinCIS architecture, we concentrate on the QNS by first presenting the subscription model, then the overall design of the service, explaining the mechanisms it provides to handle scalability issues.

5.1 Background: the DaQuinCIS project

The main motivation behind the DaQuinCIS [1] project is addressing data quality issue that arise when considering the cooperation among several organizations within a CIS. DaQuinCIS [70, 98] is a project spanning the areas of research of information systems, databases and distributed systems. In this section, we present the background of DaQuinCIS, describing how this joint research effort led to the design of a service-based distributed architecture for data quality management.

5.1.1 Data Quality in Cooperative Information Systems

A *Cooperative Information System (CIS)* is a large scale information system that interconnects various systems of different and autonomous organizations,

[1] “DaQuinCIS - Methodologies and Tools for Data Quality inside Cooperative Information Systems” (`http://www.dis.uniroma1.it/~dq/`) is a joint research project carried out by DIS - Università di Roma “La Sapienza”, DISCo - Università di Milano “Bicocca” and DEI - Politecnico di Milano.

geographically distributed and sharing common objectives [31]. As an example, a set of public administrations which need to exchange information about citizens and their health state in order to provide social aids, is a cooperative information system derived from the Italian *e*-Government scenario [8].

In the remainder of this chapter, the following definition of CIS is considered:

A cooperative information system *is formed by a set of organizations* $\{ Org_1, \ldots, Org_n \}$ *that cooperate through a communication infrastructure* $\mathbb{N}$*, which provides software services to organizations as well as general connectivity. Each organization* Org_i *is connected to* $\mathbb{N}$ *through a gateway* G_i*, on which software services offered by* Org_i *to other organizations are deployed.* A user *is a software or human entity residing within an organization and using the cooperative system.*

Among the different resources that are shared by organizations, data are fundamental; in real world scenarios, an organization $\mathcal{A}$ may not request data from an organization $\mathcal{B}$ if it does not "trust" $\mathcal{B}$ data, i.e., if $\mathcal{A}$ does not know that the quality of the data that $\mathcal{B}$ can provide is high. As an example, in an *e*-Government scenario in which public administrations cooperate in order to fulfill service requests from citizens and companies [8], administrations very often prefer asking citizens for data, rather than other administrations that have stored the same data, because the quality of such data is not known. Therefore, lack of cooperation may occur due to lack of quality certification. Uncertified quality can also cause a deterioration of the data quality inside single organizations. If organizations exchange data without knowing their actual quality, it may happen that data of low quality spread all over the CIS.

On the other hand, CIS's are characterized by high data replication, i.e., different copies of the same data are stored by different organizations. As an example, in an *e*-Government scenario, the personal data of a citizen are stored by almost all administrations. From a data quality perspective this is a great opportunity: improvement actions can be carried out on the basis of comparisons among different copies, in order either to select the most appropriate one or to reconcile available copies, thus producing a new improved copy to be notified to all interested organizations. However, in such scenarios, the different organizations can provide the same data with different quality levels; thus any user of data may appreciate to exploit the data with the highest quality level, among the provided ones.

Therefore only the highest quality data should be provided to users, limiting the dissemination of low quality data. Moreover, the comparison of the gathered data values might be used to enforce a general improvement of data quality in all organizations.

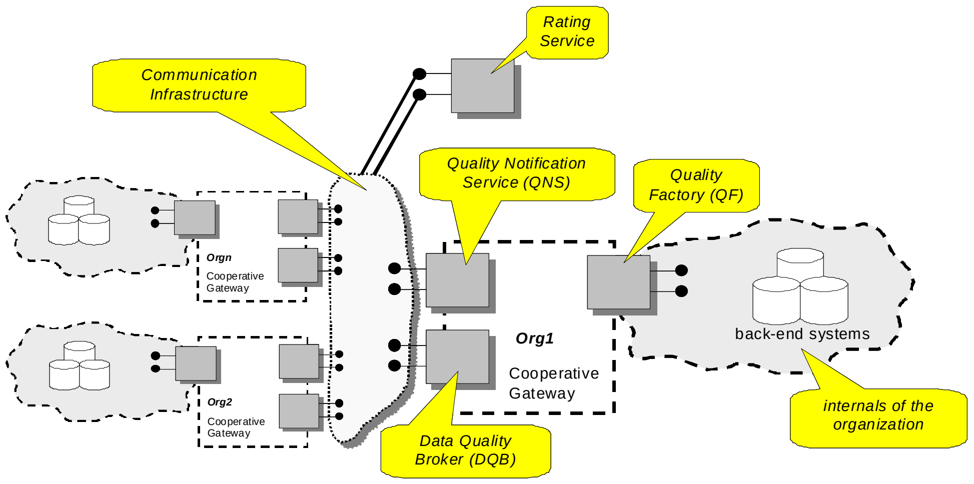
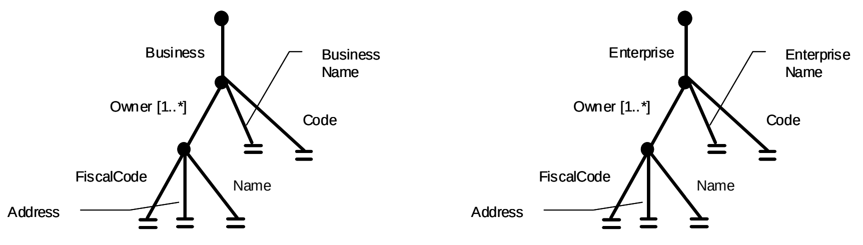


Figure 5.1: The DaQuinCIS architecture

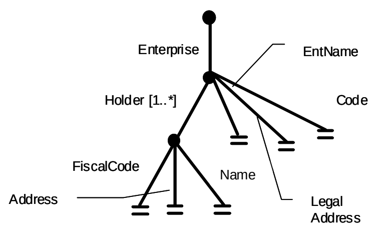
5.2 Managing Data Quality in CIS's: The DaQuinCIS Architecture

Exploiting data replication to improve the overall quality of data in the system requires on one hand the development of specific methodologies and models, and on the other hand the design of proper software tools. In the context of the DaQuinCIS project, we proposed an architecture for the management of data quality in CIS's. This architecture allows for the diffusion of data and related quality and exploits data replication in order to improve the overall quality of cooperative data. Each organization offers services to other organizations on its own cooperative gateway, and also specific services to its internal back-end systems. Therefore, cooperative gateways interface both internally and externally through services. Moreover, the communication infrastructure itself offers some specific services. Services are all identical and peer, i.e., they are instances of the same software artifacts, and act both as servers and clients of the other peers depending of the specific activities to be carried out. The overall DaQuinCIS architecture is depicted in Figure 5.1.

The DaQuinCIS project spans several areas of expertise, from distributed computing to information systems, to data integration. A complete description of the project would involve concepts that are out of the scope of this thesis. Then, for the sake of completeness and to better frame our contribution, we give only a quick overview of the DaQuinCIS architectural components. We first briefly present the data model underlying the architecture, as it is the basis of the subscription model we introduced in the QNS.



(a) Local schema of INAIL (b) Local schema of INPS



(c) Local schema of CoC

Figure 5.2: Example of local schemas for organizations in a CIS

5.2.1 The D^2Q model

The DaQuinCIS architecture requires all cooperating organizations in a CIS to export data and quality data according to a common model, referred to as *Data and Data Quality (D^2Q) model*. The model includes the definitions of *(i)* constructs to represent data, *(ii)* a common set of data quality dimensions, *(iii)* constructs to represent them and *(iv)* the association between data and quality data.

Data quality dimensions are properties of data such as correctness or degree of updating. The model exploits four dimensions gathered from the literature to give a complete, unambiguous and simple description of the quality of data. The D^2Q model adopts an XML data view: an XML Document is a set of data items, and an Document Type Definition (DTD) is the schema of such data items, consisting of *data* and *quality classes*. In particular, a D^2Q XML document contains both application data, in the form of a D^2Q *data graph*, and the related data quality values, in the form of four D^2Q *quality graphs*, one for each quality dimension. Specifically, nodes of the D^2Q data graph are linked to the corresponding ones of the D^2Q quality graphs through links. Organizations in the CIS export local schemas as D^2Q *XML DTD's*. Readers can refer to [98] for a detailed description of the model.

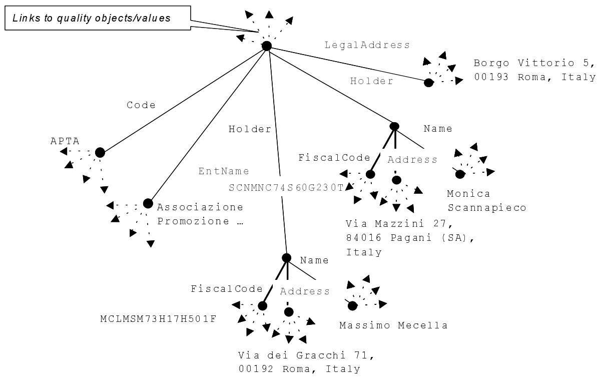


Figure 5.3: Example of a D^2Q data graph

Finally, we briefly revise a running example, also described in [98] for the DaQuinCIS project, that we will use as an example for the QNS. Figure 5.2 shows three local schemas for three different organizations in a same CIS[2]. Conceptual schemas are represented according to a hierarchical graph-based structure that can be realized through DTD's. Local schemas are represented as edge-labelled graphs, with edge labels standing for conceptual elements names.

In Figure 5.3, the D^2Q data graph of the running example is shown: an object instance `APTA` of the data class `Enterprise` is considered. The data class has `Code` and `EntName` as properties of basic types and a property of type `set-of < Holder >`; the data class `Holder` has all properties of basic types.

5.2.2 Architecture Description

In the following, we provide a brief description of each architectural component in the DaQuinCIS architecture (Figure 5.1).

In order to produce data and quality data according to the D^2Q model, each organization deploys on its cooperative gateway a **Quality Factory** service that is responsible for evaluating the quality of its own data. In practice, the Quality Factory encapsulates the method for evaluating the quality of data hosted inside an organization. Such methods are specific for each organization: imposing a common evaluation method would violate the autonomy of

[2]The example is a simplified scenario derived from the Italian Public Administration setting. Specifically, three agencies are considered: the Social Security Agency, referred to as INPS (Istituto Nazionale Previdenza Sociale), the Accident Insurance Agency, referred to as INAIL (Istituto Nazionale Assistenza Infortuni sul Lavoro), and the Chambers of Commerce, referred to as CoC (Camere di Commercio). Interested readers can refer to [2] for a precise description of the real-world scenario

each organization, that is one basic feature of CIS's. The design of the Quality Factory has been addressed in [17].

The **Data Quality Broker** poses, on behalf of a requesting user, a data request over other cooperating organizations, also specifying a set of quality requirements that the desired data have to satisfy. Different copies of the same data received as responses to the request are reconciled and a best-quality value is selected and proposed to organizations, that can choose to discard their data and to adopt higher quality ones; this process leads to an overall quality improvement inside the cooperative system. If the requirements specified in the request cannot be satisfied, then the broker initiates a negotiation with the user that can optionally weaken the constraints on the desired data.

The Data Quality Broker is in essence a peer-to-peer data integration system [65] which allows to pose quality-enhanced query over a global schema and to select data satisfying such requirements. The Data Quality Broker is described in [69].

The **Quality Notification Service** is a publish/subscribe engine, deployed as a peer-to-peer system, used as a quality message bus between services and/or organizations. More specifically, it allows quality-based subscriptions for users to be notified on changes of the quality of data. For example, an organization may want to be notified if the quality of some data it uses degrades below a certain threshold, or when high quality data are available. Similarly to what is offered by the Data Quality Broker, also the function of the QNS leads to the general improvement of the data in the CIS. The Quality Notification Service is described in Section 5.3.

The **Rating Service** associates trust values to each data source in the CIS. These values are used to determine the reliability of the quality evaluation performed by organizations. The Rating Service is a centralized service, to be provided by a third-party organization. The design of the Rating Service is addressed in [96].

5.3 The Quality Notification Service

The Quality Notification Service (QNS) is a pub/sub service used to inform interested users when changes in quality values occur within the CIS. A user willing to be notified for quality changes subscribes to the Quality Notification Service by submitting the features of the events to be notified for, through a specific-purpose content-based subscription language. When a change in quality happens, an event is published by the Quality Notification Service i.e., all the users which have a consistent subscription receive a notification.

The Quality Notification Service can be used in the CIS to control the quality of critical data, e.g., in order to keep track of its quality changes and

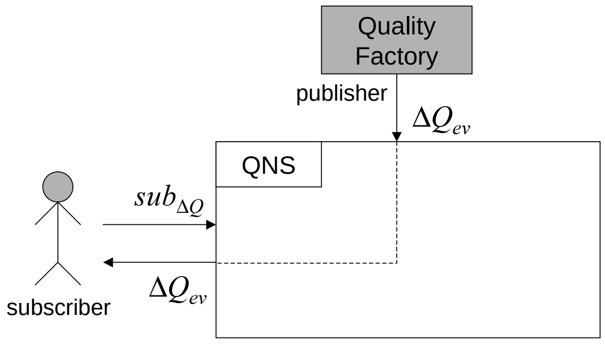


Figure 5.4: Quality Notification Service

be always aware when quality degrades under a certain threshold, which makes it no longer suited for the use it is devoted to. The Quality Notification Service can also be exploited by other architectural components to maintain updated the information they use to perform their services.

In this section we first introduce the QNS specification, i.e., *(i)* the language accepted by the Quality Notification Service to let users specify the set of notifications they are interested in, and *(ii)* the form of quality change notifications received by subscribed users; then we motivate and detail the internal architecture of the service.

5.3.1 Specification

Figure 5.4 illustrates a high-level depiction of the Quality Notification Service, and its relationships with users and with the Quality Factory. A *quality change* occurs within an organization each time the value of a given quality dimension of a property varies[3]. Quality changes are managed by the Quality Factory of each specific organization in a way that depends on the internal policies and infrastructures of the organization. Methods for evaluating quality of data have been largely studied in the research area of information quality, a survey of which can be found in [112].

Upon each quality change the Quality Factory reports to the Quality Notification Service the data object involved in the change (according to the data representation of the global schema) along with the associated quality data. As aforementioned, the Quality Notification Service is thus in charge of notifying the new data to all interested users, according to the values of the quality data.

[3]Let us remark that each attribute has an associated set of quality dimensions.

Subscription language. The Quality Notification Service subscription language has to allow quality-based subscriptions where quality dimensions are constrained by threshold values. In order to provide users with a flexible subscription language, the subscription granularity should allow to express interest in quality changes of sets of data objects. As examples, a user could be interested in monitoring accuracy changes of enterprises located in Rome whose records are stored throughout the whole CIS, whether another user could wish to subscribe for currency changes of a specific enterprise whose records are stored in a specific organization. This can be obtained with a content-based subscription model. However, with respect to a typical content-based model, our subscription model must deal with the hierarchical nature of the D^2Q model, providing a slightly different semantics for what concerns the selection of the data objects.

Therefore, a generic Quality Notification Service subscription is a triple in the form:

$$sub_{\Delta Q} = \langle \delta[:\mathcal{S}], expr_D, expr_Q \rangle$$

where:

- ❒ δ is a data class expressed on the global schema;
- ❒ $\mathcal{S}$ is an optional set of sources for the data class δ, i.e., organizations containing data objects belonging to δ according to the mapping rules;
- ❒ $expr_D$ is an expression used to specify the set of data objects of class δ whose quality changes are relevant to the user. It is a conjunction of constraints on some properties of δ. A constraint is a triple $\langle p, op, v \rangle$ where p is a property of δ, op is an operator (e.g. $=, <, >, ...$) depending on the type of p, and v is a (possibly empty) value of the same type of p;
- ❒ $expr_Q$ is an expression used to select notifications for quality changes occurred on data objects selected by $\langle \delta[:\mathcal{S}], expr_d \rangle$ according to the values of the quality data. Even this expression is a conjunction of constraints, i.e., a triple $\langle p.qd, op, t \rangle$ where $p.qd$ is a quality dimension of property p of data class δ, op is a comparison operator ($=, <, >, \leq, \geq$), and v is a numeric value ranging from 0 to 1;

At least δ must be specified, while other parameters can be left empty (i.e., set to $\perp$)[4].

[4]For the sake of simplicity, we omit to present aspects related to user unsubscriptions.

Once a user has subscribed for some notifications, he will receive them in the same form they are produced by quality factory within each organization, that is described below.

Notifications of quality changes. The Quality Factory component of each organization pushes events to the Quality Notification Service in the following form:

$$\Delta_{Q_{ev}} = \langle \delta,\ d,\ \{\ldots \langle qd,\ v\rangle_i \ldots\}\ \rangle$$

where δ is a data class of the global schema, d is a data object of class δ, and the last element is a set of pairs $\langle qd,\ v\rangle$, where qd is a quality dimension and v is the value of the associated quality data, that express the new values of the quality attributes of data object d.

Subscription Matching and Containment. As aforementioned, users will receive only those notifications that *match* their subscriptions. A notification $N = \langle \delta_N,\ d_N,\ \{\ldots \langle qd,\ v\rangle_i \ldots\}\ \rangle$ produced by a source s *matches* a subscription $S = \langle \delta_S[\colon \mathcal{S}], expr_D, expr_Q\rangle$ *iff*:

1. the data classes of N and S coincide, i.e. $\delta_N = \delta_S$, and if $\mathcal{S}$ is specified (i.e., $\mathcal{S} \neq \perp$) then $s \in \mathcal{S}$;
2. the data object d_N satisfies $expr_D$ of S;
3. the set $\{\ldots \langle qd,\ v\rangle_i \ldots\}$ satisfies $expr_Q$ of S.

Notification matching against a set of subscriptions can be efficiently evaluated using well known algorithms, e.g., [1, 80].

We now introduce the concept of *containment* between subscriptions. As shown in the following, subscription containment is useful in order to reduce both the use of memory and the network traffic due to the services provided by the Quality Notification Service. Intuitively, a subscription S_1 contains another subscription S_2 if all notifications that match S_2 also match S_1. More formally, given two subscriptions $S_1 = \langle \delta_1[\colon \mathcal{S}_1], expr_{D_1}, expr_{Q_1}\rangle$ and $S_2 = \langle \delta_2[\colon \mathcal{S}_2], expr_{D_2}, expr_{Q_2}\rangle$, then S_1 *contains* S_2 *iff*:

1. δ_1 is an ancestor node of δ_2 in the D^2Q acyclic graph and if both $\mathcal{S}_1 \neq \perp$ and $\mathcal{S}_2 \neq \perp$ then $\mathcal{S}_2 \subseteq \mathcal{S}_1$;
2. for each data object d that satisfies $expr_{D_2}$, d also satisfies $expr_{D_1}$;
3. analogously, for each set $\{\ldots \langle qd,\ v\rangle_i \ldots\}$ that satisfies $expr_{Q_2}$, also $expr_{Q_1}$ is satisfied by $\{\ldots \langle qd,\ v\rangle_i \ldots\}$.

It is possible to show that subscription containment is a transitive relationship among subscriptions, i.e. if S_1 contains S_2 that contains S_3 then S_1 contains S_3. Furthermore, it is also possible to show that checking the containment of a subscription within another requires polynomial time[5].

Quality Notification Service Running Examples. In order to illustrate the defined language and the notions of matching and containment, we now present some examples based on the global schema of the running example.

Table 5.1: Examples of subscription to Quality Notification Service

#	**Subscription**
S_1	$\langle \mathit{Enterprise/Holder}, \bot, \bot \rangle$
S_2	$\langle \mathit{Enterprise/Holder}, \bot, \mathit{Name.Accuracy} \leq \texttt{medium} \rangle$
S_3	$\langle \mathit{Enterprise/Holder}, \mathit{FiscalCode} = \text{``}MCLMSM73H17H501F'', \ldots$
	$\ldots \mathit{Name.Accuracy} \leq \texttt{medium} \rangle$
S_4	$\langle \mathit{Enterprise/Holder} : \{INPS, INAIL\}, \mathit{FiscalCode} = \text{``}MCLMSM73H17H501F'', \ldots$
	$\ldots \mathit{Name.Accuracy} \leq \texttt{medium} \rangle$

Consider the set of example subscriptions in Table 5.1. Subscription S_1 refers to each quality change notifications for data objects belonging to the class `Holder` (son of class `Enterprise`), from any data source in the system. The following subscriptions limit the set of interesting quality change notifications. In particular, S_2 subscribes for notifications of quality changes in the `Name` attribute of objects of class `Holder`, when the accuracy of this attribute falls under or is equal to `medium`. Subscription S_3 further restricts the set of data objects of interest to holders whose fiscal code is equal to a specific value. Finally, subscription S_4 is the same as the previous one, but involves only data objects stored in two specific sources.

It is easy to see that subscription S_2 is contained in subscription S_1 and contains S_3 that in turn contains S_4. Therefore S_1 contains all others.

Concerning matching, a valid quality change event generated by a Quality Factory might be:

$$\langle \mathit{Enterprise/Holder}, \mathit{Name} = \text{``}M.\ Mecalla''\mathit{Address} = \\ \text{``}Via\ dei\ Gracchi\ 71,\ Roma''\mathit{FiscalCode} = \\ \text{``}MCLMSM73H17H501F'',\ \{\langle \texttt{name.Accuracy} = \mathit{low} \rangle\} \rangle$$

The afore event is received by Quality Notification Service that notifies all interested users. The notification has the same form of the event pushed by

[5]More precisely, if the complexity of checking if a constraint is contained within another is constant, being n the sum of the number of constraints contained in two subscriptions, then the check costs $O(n^2)$.

QF. In the case of the example subscriptions given in Table 5.1, it is easy to see that the notification matches S_1, S_2, and S_3. Matching with subscription S_4 depends on the organization in which the quality change occurs.

5.4 QNS Design

The Quality Notification Service is implemented and deployed as a distributed service. Each organization hosts an independent instance of the Quality Notification Service that adopts the architecture described in this section. A Quality Notification Service instance accepts subscriptions only from users inside the organization it resides in (i.e. its *local users*) and receives notifications from the local Quality Factory. Quality Notification Service instances communicate among each other in a peer-to-peer fashion. That is, a Quality Notification Service [6] acts as a *producer* of information when it has to propagate to other Quality Notification Services the notifications produced by the Quality Factory of its organization. On the other hand, QNS acts as a *consumer* of information when it receives notifications produced in other organizations' Quality Notification Services, to be dispatched to its local users. Clearly, a Quality Notification Service can be a producer only for data classes of the global schema for which its organization is a source.

5.4.1 Overview and Motivations

The design of the internal architecture of the Quality Notification Service has to face two main problems: *(i)* to cope with the heterogeneity of platforms and network infrastructures building the system and *(ii)* to scale to the large size we can expect in a CIS context.

Concerning heterogeneity, a standard technological solution, i.e. Web Services [79], is exploited for the communications among Quality Notification Service instances in order to achieve independence from the specific technological platform of each organization. In practice, the various Quality Notification Service instances realize a distributed protocol for notification routing by exchanging SOAP [27] messages over the HTTP protocol.

Concerning scale, the high number of possible users, each possibly issuing several subscriptions, and the high rate of notifications possibly produced by the Quality Factories are factors that could impact the performance of the system, in terms of computational resources and network bandwidth. In particular, the high number of subscriptions that could be issued throughout the whole system, imposes a high memory and computational overhead for stor-

[6] In the following, for simplicity we refer to a Quality Notification Service instance as "a Quality Notification Service " whenever this is not confusing.

ing and matching each notification, while possible frequent notifications may generate a large network traffic.

Both problems of heterogeneity and scale are faced through a layered architecture, in which a layer (namely, the *External Diffusion Layer, EDL*) deals with diffusing subscriptions and notifications among all organizations, while another layer (namely, the *Internal Interface Layer, IIL*) deals with handling subscriptions and notifications for all local users. The subscriptions of local users in an organization are stored inside the IIL. That is, each Quality Notification Service instance "knows" only its local users' subscriptions but has only a *partial* knowledge about the subscriptions issued by users of other organizations. Knowledge about non-local subscriptions is propagated *selectively* by exploiting containment relations, as we detail in the following section, limiting memory consumption and network traffic.

5.4.2 Merge Subscriptions

When a Quality Notification Service acts as a producer, it triggers a notification routing algorithm involving all the consumer Quality Notification Services. As we pointed out in Chapter 2, the trivial solution for realizing this routing algorithm under a complete absence of "knowledge" about subscriptions, is to blindly flood all the notifications to all other Quality Notification Services, in order to surely reach all interested users. This obviously would cause much unnecessary network traffic as each notification would be received also by Quality Notification Services with no interested local users.

To avoid this, the EDL of the Quality Notification Service in the generic organization O_i maintains a particular subscription, namely the *merge subscription*, denoted as $\mathcal{M}_i$. $\mathcal{M}_i$ contains all and only the subscriptions of all the local users of O_i and is computed from the union of the latter. For example the merge of subscriptions S_2, S_3 and S_4 in the examples of Table 5.1 is the subscription S_2 itself. $\mathcal{M}_i$ allows to receive all and only notifications matching some subscriptions issued by some users of O_i.

The merge subscription of each consumer organization is sent to all the producers of notifications possibly matching the merge subscription itself. On the consumer side, the schema mapping is exploited to determine the set of possible producers for a subscription. This allows for further reducing the Quality Notification Service instances involved in the communication with a given consumer only to those that can actually generate *interesting* notifications.

Upon a new notification, a Quality Notification Service acting as a producer has to check whether the notification matches the merge subscription of some other Quality Notification Services. Let us show this with the example depicted in Figure 5.5. Consider a producer Quality Notification Service,

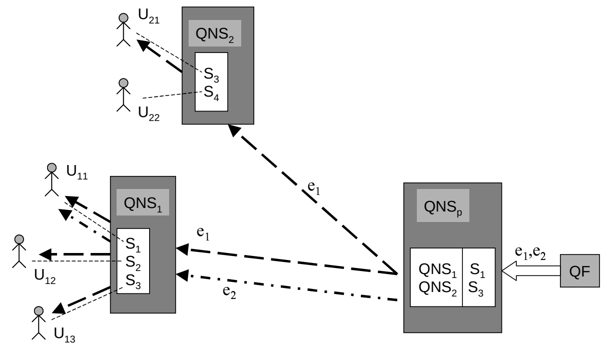


Figure 5.5: Exploiting Merge Subscriptions

QNS_{CoC}, and two consumers QNS_{INPS} and QNS_{INAIL}. Suppose QNS_{INPS} manages three users, respectively interested in subscriptions S_1, S_2 and S_3 of example in Table 5.1, while QNS_{INAIL} manages two users, interested in subscription S_3 and S_4. The merge subscriptions for QNS_{INPS} and QNS_{INAIL} are respectively subscription S_1 and S_3. These are stored in QNS_{CoC}. Now consider the two following quality changes events generated in CoC for the data class *holder*:

$$e1: \langle Enterprise/Holder, Name = \text{``}M.\ Mecalla'' Address = \text{``}Via\ dei\ Gracchi\ 71, Roma'' FiscalCode = \text{``}MCLMSM73H17H501F'',\ \{\langle \texttt{name.Accuracy} = low\rangle\}\ \rangle$$

$$e2: \langle Enterprise/Holder, Name = \text{``}M.\ Scannapieco'' Address = \text{``}Via\ Mazzini\ 27,\ Pagani\ (SA)'' FiscalCode = \text{``}SCNMNC74S70G230T\text{''},\ \{\langle \texttt{name.Accuracy} = medium\rangle\}\ \rangle$$

$e1$ matches S_1, S_2 and S_3, while $e2$ matches only S_1. Due to the afore propagation of merge subscriptions, QNS_{CoC} can avoid the sending of $e2$ to QNS_{INAIL} as it can state from the merge subscriptions that QNS_{INAIL} has no users interested in $e2$. On the other hand, $e1$ is sent to both consumers Quality Notification Services, as they all have users interested in it. However, note that $e2$ is forwarded only to user U_{11} in QNS_{INPS} and $e1$ is not forwarded to user U_{22} in QNS_{INAIL}. However, the additional matching check required to determine the users of a notification is handled by consumer Quality Notification Services, independently from the producer.

It is easy to devise that subscription traffic is highly reduced due to the use of merge subscriptions. Indeed, only changes in subscription that provoke a change in a merge subscription of an organization have to be propagated to producers. For example, if a new user in QNS_1 subscribes to subscription S_2, this is not communicated to QNS_p because it does not affect the merge subscription, that remains S_1.

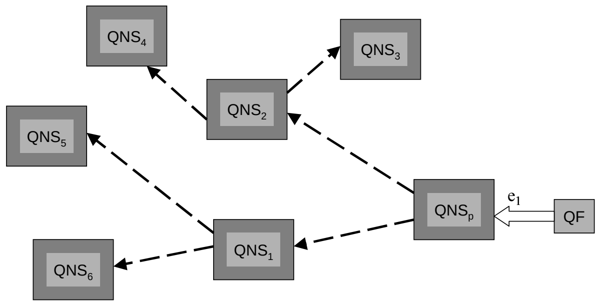


Figure 5.6: Diffusion Tree Example

In overall, the use of merge subscriptions requires an additional computational step when a new subscription and a new notification are issued. As valuable counterpart, the step allows to let flow between consumer and producer organizations only the *necessary* subscriptions and notifications, cutting off all the "useless" inter-organization network traffic.

Nevertheless, maintaining such a high number of SOAP connections could be enough to overload some Quality Notification Services. In the following we describe a mean to reduce the probability that this event occur.

5.4.3 Diffusion Trees

If a large number of Quality Notification Service consumers is interested in notifications coming from the same producer Quality Notification Service, the latter could be forced to open a large number of concurrent network connections to reach all of them. In the presence of a high rate of notifications, the scalability of the system could result compromised. To reduce the impact of this phenomena, each producer Quality Notification Service, once determined all the consumers of a notification, does not directly send it to them, but it rather calculates a diffusion tree, rooted at the producer and spanning all the consumers. The fan out of each node of the tree is evaluated on the basis of information about the network connectivity capabilities of each network node.

Each notification is thus sent from a node of the tree to its sons along with the information about the subtree of further consumers interested in the notification (see Figure 5.6). Therefore, Quality Notification Services at intermediate levels of the tree act as forwarders of the notification for the lower levels. That is, upon receiving a notification, they also send it again to the Quality Notification Services at the immediate lower level in the tree. This approach enables (i) to limit the number of concurrent network connections managed by a producer and (ii) to distribute the load of notification propagation among all consumers according to their capabilities.

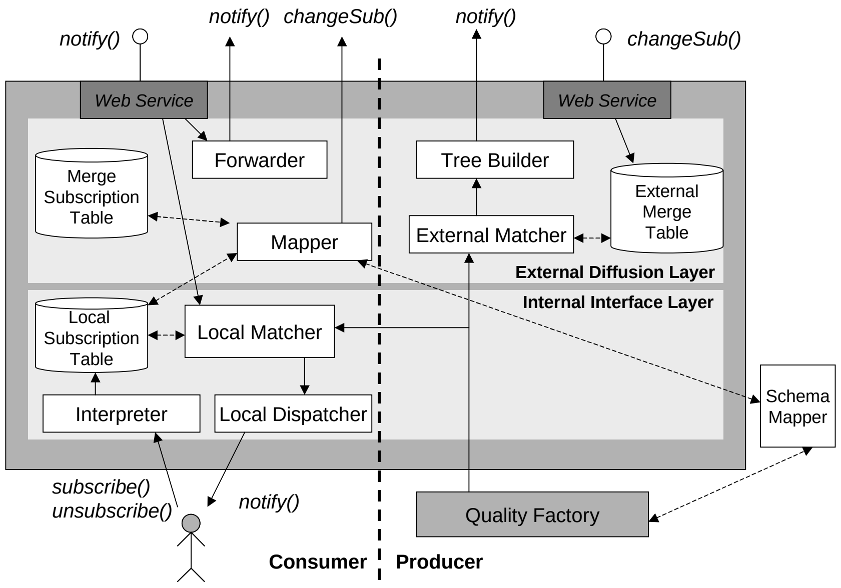


Figure 5.7: Quality Notification Service

5.4.4 Quality Notification Service Internal Architecture

Figure 5.7 depicts the internal architecture of a generic Quality Notification Service instance, and shows also the internal fine-grained functional components of each of the Quality Notification Service layers. In the remainder of this section we detail these components.

Internal Interface Layer (IIL). The IIL consumer side comprises the following components:

❒ *Interpreter*: receives subscriptions from local users, interprets the subscription language, and stores them into the *Local Subscription Table*, i.e. a data structure storing all the subscriptions of all local users of an organization;

❒ *Local Matcher*: receives quality change notifications from the quality factory and from the EDL, and then matches them against the Local Subscription Table in order to return notifications to interested local users;

❒ *Local Dispatcher*: implements the dispatching mechanisms used to actually send notifications to local users.

The implementation of the subscription and dispatching mechanisms depend on the specific internal communication infrastructure of each organization, i.e. the Interpreted and the Local Dispatcher can be customized for each Quality Notification Service instance taking into account specific technological issues (e.g. pre-existence of a publish/subscribe middleware);

On the producer side, the IIL receives quality change events from the local Quality Factory of the organization. The data is first checked inside the organization via the Local Matcher to be dispatched to interested local users. The data is also transferred to components of the EDL to propagate it outside the organization.

External Diffusion Layer (EDL). As aforementioned, this layer is responsible for inter-organization notification diffusion. We explained in section 5.4 that the communication among different organization is based on Web Services technology. This means that each EDL instance exposes a Web Service interface, containing two methods, namely `notify()` (on the consumer side) and `changeSub()` (on the producer side), that are used to receive from other Quality Notification Services a notification and a change of merge subscription, respectively.

The producer side of the EDL of a Quality Notification Service instance comprises the following components:

❒ *External Matcher*: it checks the Quality Notification Service instances to which a notification has to be sent, by matching the notification against the merge subscriptions of other Quality Notification Services. The *External Merge Table* data structure maintains the merge subscriptions of consumer organizations, received from the Merger component of their Quality Notification Service (see below). As a consequence, such organizations are all and only the possible consumers (i.e. they have at least one subscribed user) for data classes for which O_i is a producer. Figure 5.5 shows the External Merge Table for QNS_{CoC}.

❒ *Tree Builder*: Determines the actual diffusion path followed by notifications and starts the notification diffusion. Notifications are propagated by invoking the `notify()` method on consumer Quality Notification Services. Each invocation contains also the specification of the remaining parts of the diffusion tree to which the notification has to be forwarded by the receiver, as shown in Figure 5.6.

The consumer side of the EDL of a Quality Notification Service instance comprises the following components:

❒ *Merger*: determines the merge subscription for the organization, when a subscribe/unsubscribe request occurs. A data structure, namely the

Merge Subscriptions Table, maintains the merge subscription of the organization as seen by each producer Quality Notification Service [7]. The Merger sends updates to producers whenever the merge subscription changes. As aforementioned, producers are identified using the *Schema Mapper*.

- ❒ *Forwarder*: responsible for sending the notification to all the Quality Notification Services at lower levels of the diffusion tree, by invoking their `notify()` method.

Let us conclude this section by pointing out the motivations that led us to choose this design rather than one straightforwardly based on an application-level network, where the EDL has the role of a broker, such as the one described in Chapter 4. This solution, after being first took into consideration, was finally rejected as we did not believe it appropriate for this scenario. This decision was driven by the following observations: *(i)* differently from the general model we considered in Chapter 4, where each broker could be the producer for any notification, in this case, due to the schema mapping, each Quality Notification Service can produce only a subset of the overall notifications. Each subscription change has to be communicated only to a small subset of other Quality Notification Services; *(ii)* in an average CIS, the overall number of organizations (and subsequently of Quality Notification Services) is expected not to be higher than a few hundreds. In other words, the main scalability metric is the number of subscriptions/notifications exchanged rather than the size of the system.

In overall we can say that the existence of other items of the DaQuinCIS framework (in particular, of the D^2Q model, of the Schema Mapper, and of the Quality Factory) has made possible these simplifications in the Quality Notification Service design, that would not have been feasible in a completely general model. In essence, this is what makes the difference between the Quality Notification Service and a general-purpose pub/sub middleware.

5.5 Implementation and Simulation

The implementation of the QNS is currently has been written in Java, exploiting the JAX-RPC API [73] for the development of the web services interaction among the EDLs. Currently, we are including simple straightforward algorithms [75] for evaluating containment and matching of subscriptions. A

[7]Let us remark that a merge subscription is, in the general case, obtained from the union of a set of user subscriptions that do not satisfy containment. Therefore a merge subscription can be represented as a set of user subscriptions, which are stored in the Merge Subscription Table.

future version will include specific optimizations of these algorithms for data represented in XML [66].

In the following we present a performance study of the QNS, with 10 QNS instances running and exchanging messages. The objective of the simulations is to evaluate the number of SOAP messages and the subscription table size saved thanks to the merge subscriptions technique, with respect to a trivial solutions where all the subscription changes are sent and stored within all the QNSs. Simulation scenarios were generated randomly over an example schema for subscriptions and notifications (i.e., a weather report application). We also considered a particular mapping for the schema where each organization hosts only a part of the global schema. Subscriptions follow a Zipf distribution for each attribute (including quality dimensions) while notifications follow a uniform distribution for each attribute.

Figures 5.8(a) and 5.8(b) report the effect of merge subscriptions respectively on the network traffic and on the average size of external merge tables of each QNS instance. Plots show the percentage reduction with respect to the trivial solution, respectively with and without considering the schema mapping (in the latter case, all organizations hold the entire schema). Network traffic is measured in terms of overall number of SOAP messages sent, while merge table size is measured in terms of number of subscriptions. Improvements due to the use of merge subscriptions are clear from the Figure, showing in both cases a reduction that reaches the 75% of the values obtained in the basic case. To focus on the effect of the schema mapping, Figure 5.9 shows a distribution of the sizes of subscription tables in all the organizations (each curve corresponds to a different organization).

Finally, Figure 5.10 shows the effect of merge subscription on the SOAP traffic due to notifications. In the trivial case a notification is sent once for each distinct subscriber, while when using merge subscription, a notification is sent from a QNS to another only once for all the subscribers served by it. Plots were produced changing the number of publication issued while keeping a fixed number of subscriptions (100 per organization). The two curves plot the number of notifications actually delivered to subscribers (dashed line) and the ones sent from one QNS to another, when merge subscriptions are used. In practice, this is the comparison between the SOAP traffic due to notifications obtained in the basic solution and the optimized one, showing that merge subscriptions allow to dramatically reduce the SOAP traffic in the system.

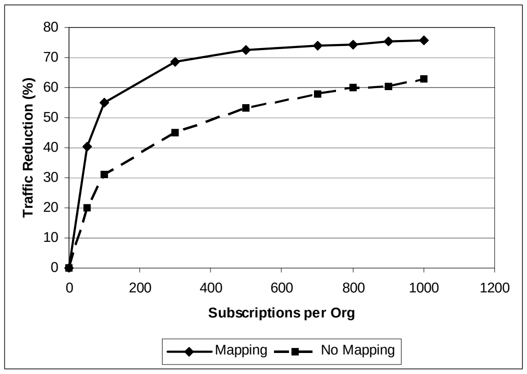
5.6 Related Work

Only few systems and architectures for the management of data quality in distributed systems have been proposed in the literature [114, 97, 63]. None of them includes a pub/sub-like mechanism such as the QNS. Descriptions of practical experiences with pub/sub system can be found in [11, 81, 35]. Two example applications exploiting a pub/sub system are also presented in [76] as a case study for REBECA. However, the QNS is rather a domain-specific application than an actually deployed system, though we plan to experiment its usage on real-world data in future work. Differently from our approach, where the design of the QNS is tailored to the applicative domain, in all the aforementioned work pub/sub is used as a general middleware service.

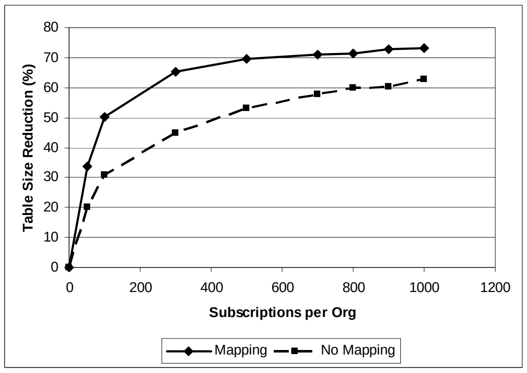
Finally, for what concerns the specific design of the QNS, it can resemble the two-level daemon architecture in TIB/RV. The main difference is that we are considering a content-based model instead of the simple topic-based model in TIB/RV and this makes more challenging managing subscription traffic in the higher level of the hierarchy. Moreover, TIB/RV does not consider the integration issue, that we tackled by realizing an architecture based on web services.

5.7 Concluding Remarks

The DaQuinCIS project is an interesting meeting point of different research areas. In particular, under the point of view of large-scale distributed systems, we explored which challenges could from the data quality management and from the cooperative context to the problem of designing a scalable pub/sub service. The result is the design of the QNS, in which we adapted some ideas form the pub/sub solutions we considered in previous Chapters. In particular, merge subscriptions are a coarse-grained form of content-based routing while with diffusion trees we realized a simplified form of self-organizing system.



(a)



(b)

Figure 5.8: Effect of Merge Subscriptions on Subscription Traffic and Subscription Tables Size

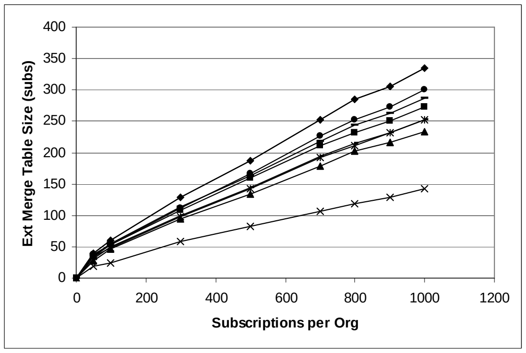


Figure 5.9: Subscription Table Size for each Organization

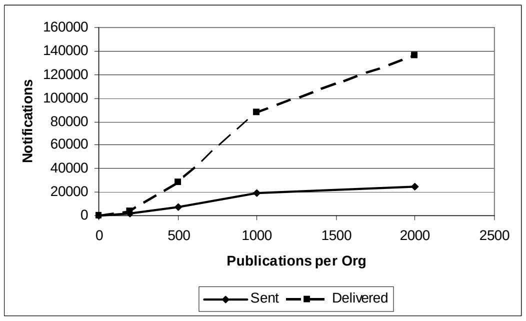


Figure 5.10: Effect of Merge Subscriptions on Notification Traffic

Chapter 6

Conclusions

The publish/subscribe paradigm is largely recognized as one of the most effective way to realize flexible and scalable large-scale distributed applications. The advantages of the paradigm and its peculiar features have been discussed several times and now are consolidated concepts. However, realizing an Internet-scalable Notification Service still remains a big challenge, though it has been tackled in the past by several researchers. Starting from this observation, in this thesis we proposed a set of contributions aiming, on one hand, at identifying and classifying the key ideas underlying the pub/sub paradigm, on the other, to suggest new problems and new directions in which the research on pub/sub can be driven.

6.1 Contributions and Future Work

In the following a summary of the contributions of the thesis is given. For each of them we also envision the possible future developments.

Understanding pub/sub systems In Chapter 2 we presented the state of the art in the publish/subcribe research area. With respect to previous surveys [40, 71] we focused on a detailed analysis of the internal mechanisms of pub/sub, considering the consequence of the choice of the subscription models and architectural models and pointing out the trade-offs arising between the different solutions for routing notifications and subscriptions. We also introduced the novel concept of subscription assignment, that allows to clearly distinguish the different architectural and routing solutions adopted from early systems to more recent pub/sub incarnations.

Modelling pub/sub systems In Chapter 3 we presented two models that represent the behavior of a pub/sub system under two points of view: a computational model, capturing the exact semantics of the system and its non-deterministic nature, and an analytical model, for the evaluation of the probability that a notification is not actually delivered to subscribers, because of non-determinism. Experimental studies showed that the analytical model gives a good evaluation of this probability, given the operative parameters N, the number of participants in the Notification Service, and α, the propagation speed of updates for notifications and subscriptions inside the Notification Service. At the current state of the work we assume that the evaluation of this latter parameter is done experimentally. However, in the future work we plan to extend the model in order to more precisely characterize α, providing some analytical expression that allows evaluation starting from the deployment characteristics of the system.

Self-organizing Pub/sub Systems In Chapter 4 we presented a novel self-organization algorithm, namely SOCBR, for content-based pub/sub systems built as an application-level network of brokers. The algorithm rearranges the TCP connections among brokers with the aim of increasing the system's associativity, a metric representing commonality of interest among subscribers. We performed an extensive simulation study, realized by implementing a full content-based pub/sub system enhanced with the self-organization algorithm, running on top of the J-Sim simulator.

Experimental results show clearly that, firstly, increasing the associativity produces a reduction in the TCP hops metric when performing routing of notifications and, secondly, the self-organizing algorithm allows to reduce the overall application-level network traffic of content-based routing when the rate of notification publications is roughly ten times higher than subscriptions rate (a common situation for pub/sub systems). We show that the benefits outweigh the costs of self-organization starting from very low relative rates and reaching improvements of 30% in the overall number of TCP hops.

We also presented pbSOCBR, a network-aware version of SOCBR that limits the reconfiguration of the application-level network, by changing its topology only into one that do not spoil the average network-level latency of notification routing. Adding network-awareness to pub/sub systems without relying on an underlying peer-to-peer overlay network is a novel achievement for the pub/sub research area.

In the future work we plan to evolve our simulation prototype into a full, stand-alone pub/sub system based on the SOCBR and pbSOCBR algorithms. The basic step toward this direction is the inclusion of more efficient algorithms for the computation of associativity. Another idea in this direction could be the inclusion of an on-line clustering of subscriptions: this allow to greatly simplify content-based routing, and probably self-organization itself, by introducing the notion of groups. However, clustering obviously introduces an error, by aggregating into a same group subscriptions that are not exactly equal. Then, a notification could be routed to subscribers that are not actually interested in it, introducing some useless hops. This could raise some interesting research issues for the future work such as the evaluation of the impact of this error over the overall benefits of self-organizations and the comparison with the results presented in this thesis using a precise evaluation of similarity.

As another direction for the future work, we plan to exploit the results of the analytical model as a feedback for the self-organization process. The idea is the following: the model has to be specialized in order to calculate the value of α for a particular application-level topology. α should reflect the quality of the mapping between the application-level topology and the network-level topology. Thus, self-organization will be driven by two, independent factors: i) trying to enhance the overall associativity of the system and ii) trying at the same time not to decrease the value of α.

Pub/sub for Data Quality Notification In Chapter 5 we presented the design of a pub/sub system tailored for managing data quality issues. This work was made in the context of the DaQuinCIS project, where we propose an architecture for the management of data quality in Cooperative Information Systems. CISs are often characterized by a high degree of data replication: that is, organizations typically provide the same information with distinct quality levels and this enables providing users with data of the highest available quality. Furthermore, the comparison of data values might be used to enforce a general improvement of data quality in all organizations. The DaQuinCIS architecture allows the diffusion of data and related quality and exploits data replication to improve the overall quality of cooperative data.

In this context we focused on the design of a specific-purpose pub/sub service for data quality notification, namely the Quality Notification Service. Available pub/sub infrastructures do not allow to meet all the requirements that a QNS implementation should satisfy, in particular scaling to a large number of users and coping with platform heterogeneity, because of the generic assumptions they have been built for that do not allow for context-specific optimizations. QNS addresses both these problems through a layered architecture that (i) encapsulates the technological infrastructure specific of each organization, (ii) adopts the standard Web-service technology to implement inter-organization communications, and (iii) embeds solutions and algorithms (namely, merge subscriptions and diffusion trees) to reduce the use of physical and computational resources.

6.2 Future Directions

We conclude the thesis by proposing some other ideas for possible directions for future research in pub/sub. For example, the realization of efficient and dynamic filter-driven assignment policy is a problem that still has to be completely solved. The FDA approach in our opinion is a promising direction of research, because it can result in very efficient routing protocols and effective load balance among the brokers. However, current solutions [83, 110] still rely on too many restrictive assumptions, especially when considering a content-based subscription model.

But probably the major research challenges are related to the problem of information diffusion over mobile environments. More specifically, the open question is how to realize an efficient and scalable pub/sub dissemination in a network composed by mobile devices. Many practical problems are related to this setting (e.g. unpredictable losses in connectivity, limited power of devices, etc.) making it an exciting test-bed for researchers. Though several contributions for a mobile pub/sub already have been presented [57, 4, 26], this is probably where much of the future research will be directed.

Bibliography

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, *Matching Events in a Content-Based Subscription System*, Proceedings of The Symposium on Principles of Distributed Computing, 1999, pp. 53–61.

[2] P. Aimetti, C. Batini, C. Gagliardi, and M. Scannapieco, *The "Services To Enterprises" Project: an Experience of Data Quality Improvement in the Italian Public Administration*, Experience Paper at the 7th International Conference on Information Quality (IQ'02), Boston, MA, USA, 2001.

[3] M. Altherr, M. Erzberg, and S. Maffeis, *iBus - a software bus middleware for the java platform*, Proceedings of the International Workshop on Reliable Middleware Systems, October 1999, pp. 49–65.

[4] Emmanuelle Anceaume, Ajoy K. Datta, Maria Gradinariu, and Gwendal Simon, *Publish/Subscribe Scheme for Mobile Networks*, Proceedings of the Workshop on Principles on Mobile Computing (POMC'02), 2002.

[5] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito, *The notion of availability in publish/subscribe communication systems*, Proceedings of Workshop on Foundations of Middleware Technologies (WFoMT 2002), Irvine, CA, USA, November 2002.

[6] R. Baldoni, M. Contenti, and A. Virgillito, *The Evolution of Publish/Subscribe Systems*, Future Trends in Distributed Computing, Research and Position Papers (André Schiper and Alexander A. Shvartsman and Hakim Weatherspoon and Ben Y. Zhao, ed.), vol. 2584, Springer, 2003.

[7] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman, *An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems*, Proceedings of International Conference on Distributed Computing Systems, 1999.

[8] C. Batini and M. Mecella, *Enabling Italian* e-*Government Through a Cooperative Architecture*, IEEE Computer **34** (2001), no. 2.

[9] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach, *Exactly-once Delivery in a Content-based Publish-Subscribe System*, Proceedings of The International Conference on Dependable Systems and Networks, 2002.

[10] K. P. Birman, *The process group approach to reliable distributed computing*, Communications of the ACM **12** (1993), no. 36, 36–53.

[11] ______, *A review of experiences with reliable multicast*, Software - Practice & Experiences **9** (1999), no. 29, 741–774.

[12] K. P. Birman and R. van Renesse, *Reliable distributed computing with the isis toolkit*, IEEE Computer Society Press, 1994.

[13] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky, *Bimodal multicast*, ACM Transactions on Computer Systems (TOCS) **17** (1999), no. 2, 41–88.

[14] Andrew D. Birrell and Bruce Jay Nelson, *Implementing remote procedure calls*, ACM Transactions on Computer Systems (TOCS) **2** (1984), no. 1, 39–59.

[15] Adrian Bozdog, Robbert van Renesse, and Dan Dumitriu, *Selectcast – a scalable and self-repairing multicast overlay routing facility*, Proceedings of the First ACM Workshop on Survivable and Self-Regenerative Systems, 2003.

[16] A. Campailla, S. Chaki, E. M. Clarke, S. Jha, and H. Veith, *Efficient filtering in publish-subscribe systems using binary decision diagrams*, Proceedings of The International Conference on Software Engineering, 2001, pp. 443–452.

[17] C. Cappiello, C. Francalanci, B. Pernici, P. Plebani, and M. Scannapieco, *Data quality assurance in cooperative information systems: a multi-dimension quality certificate*, Proceedings of the ICDT'03 International Workshop on Data Quality in Cooperative Information Systems (DQCIS'03), Siena, Italy, 2003.

[18] A. Carzaniga, D. Rosenblum, and A. Wolf, *Challenges for distributed event services: Scalability vs. Expressiveness*, Engineering Distributed Objects '99, Los Angeles CA, USA, May 1999.

[19] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, *Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service*, Proceedings of the ACM Symposium on Principles of Distributed Computing, 2000, pp. 219–227.

[20] ———, *Design and Evaluation of a Wide-Area Notification Service*, ACM Transactions on Computer Systems **3** (Aug 2001), no. 19, 332–383.

[21] Antonio Carzaniga, *Architectures for an Event Notification Scalable to Wide-Area Networks*, Phd thesis, Politecnico di Milano, 1998.

[22] M. Castro, P. Druschel, A. Kermarrec, and A. Rowston, *Scribe: A large-scale and decentralized application-level multicast infrastructure*, IEEE Journal on Selected Areas in Communications **20** (October 2002), no. 8.

[23] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowston, *Exploiting network proximity in distributed hash tables*, Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo), Bertinoro, Italy, June, 2002, 2002.

[24] ———, *Exploiting Network Proximity in Peer-to-Peer Overlay Networks*, Tech. report, Technical report MSR-TR-2002-82, 2002.

[25] D. R. Cheriton and W. Zwaenpoel, *Distributed process groups in the v kernel*, ACM TOCS **3** (1985), no. 2, 77–107.

[26] M. Cilia, L. Fiege, C.Haul, A.Zeidler, and A. P. Buchmann, *Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware*, Proceedings of the Second International Workshop on Distributed Event-Based Systems (DEBS'03), 2003.

[27] World Wide Web Consortium, *Soap version 1.2 specification*, 2003.

[28] P. Costa, M. Migliavacca, G.P. Picco, and G. Cugola, *Introducing Reliability in Content-Based Publish-Subscribe through Epidemic Algorithms*, Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03), 2003.

[29] G. Cugola, E. Di Nitto, and A. Fuggetta, *Exploiting an event-based infrastructure to develop complex distributed systems*, Proceedings of the 10th International Conference on Software Engineering (ICSE '98), April 1998.

[30] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta, *The JEDI Event-Based Infrastructure and Its Application to the Development of the*

OPSS WFMS, IEEE Transactions on Software Engineering **27** (2001), no. 9, 827–850.

[31] G. De Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, M.P. Papazoglou, K. Pohl, J. Schmidt, C. Woo, and E. Yu, *Cooperative information systems: A manifesto*, Cooperative Information Systems: Trends & Directions (M.P. Papazoglou and G. Schlageter, eds.), Accademic-Press, 1997.

[32] S. E. Deering and D. R. Cheriton, *Multicast routing in datagram networks and extended lans*, ACM Transactions on Computer Systems **8** (1990), no. 2, 85–111.

[33] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, and John Larson, *Epidemic algorithms for replicated database maintenance*, Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing, 1987, pp. 1–12.

[34] C. Diot, B.N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, *Deployment issues for the ip multicast service*, IEEE Network Magazine, special issue on Multicasting (2000).

[35] Srgio Duarte, J. Legatheaux Martins, Henrique J. Domingos, and Nuno Preguia, *A case study on event dissemination in an active overlay network environment*, Proceedings of 2nd International Workshop on Distributed Event-Based Systems (DEBS'03), 2003.

[36] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov, *Lightweight probabilistic broadcast*, Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001), July 2001.

[37] P. Th. Eugster, *Type-based publish/subscribe*, Ph.D. thesis, EPFL, December 2001.

[38] P. Th. Eugster and R. Guerraoui, *Probabilistic multicast*, Proceedings of the 3rd IEEE International Conference on Dependable Systems and Networks (DSN 2002), 2002, pp. 313–322.

[39] P.Th. Eugster, P. Felber, R. Guerraoui, and S.B. Handurukande, *Event Systems: How to Have Your Cake and Eat It Too*, Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'02), 2002.

[40] P.Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec, *The Many Faces of Publish/Subscribe*, ACM Computing Surveys **35** (2003), no. 2, 114–131.

[41] P.Th. Eugster and R. Guerraoui, *Distributed Programming with Typed Events*, IEEE Software (2003).

[42] P.Th. Eugster, R. Guerraoui, and Ch.H. Damm, *On Objects and Events*, Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2001.

[43] P.Th. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie, *From Epidemics to Distributed Computing*, IEEE Computer (2003).

[44] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, *Filtering algorithms and implementation for very fast publish/subscribe*, Proceedings of the 20th Intl. Conference on Management of Data (SIGMOD 2001), 2001, pp. 115–126.

[45] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang, *Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing*, IEEE/ACM Transactions on Networking (November 1996), 784–803.

[46] E. Freeman, S. Hupfer, and K. Arnold, *Javaspaces principles, patterns, and practice*, Addison-Wesley, 1999.

[47] D. Gelernter, *Generative communication in linda*, ACM Computing Surveys **7** (1985), no. 1, 80–112.

[48] K. Gough and G. Smith, *Efficient recognition of events in distributed systems*, Proceedings of the ACSC-18, 1995.

[49] Object Management Group, *CORBA event service specification, version 1.1*, OMG Document formal/2000-03-01, 2001.

[50] ———, *The common object request broker architecture, version 3.0*, OMG Document formal/2002-12-02, 2002.

[51] ———, *CORBA notification service specification, version 1.0.1*, OMG Document formal/2002-08-04, 2002.

[52] R. E. Gruber, B. Krishnamurthy, and E. Panagosf, *The architecture of the ready event notification service*, Proceedings of The International Conference on Distributed Computing Systems, Workshop on Middleware, Austin, Texas, 1999.

[53] Gryphon Web Site, *http://www.research.ibm.com/gryphon/.*

[54] M. Guimaraes and L. Rodrigues, *A genetic algorithm for multicast mapping in publish-subscribe systems*, Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications, 2003, pp. 67–74.

[55] Antonin Guttman, *R-trees: a dynamic index structure for spatial searching*, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, ACM Press, 1984, pp. 47–57.

[56] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton, *Log-Based Receiver-ReliableMulticast for Distributed Interactive Simulation*, Proceedings of the 1995 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM 95), August 1995, pp. 328–341.

[57] Yongqiang Huang and Hector Garcia-Molina, *Publish/Subscribe in a Mobile Environment*, Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), 2001.

[58] IBM, *Websphere mq*, http://www.ibm.com/websphere, 2003.

[59] J-Sim, http://www.j-sim.org, 2003.

[60] A.-M. Kermarrec, L. Massouli, and A.J. Ganesh, *Probabilistic Reliable Dissemination in Large-Scale Systems*, IEEE Transactions on Parallel and Distributed Systems **14** (2003), no. 3.

[61] Abdelmajid Khelil, Christian Becker, Jing Tian, and Kurt Rothermel, *An epidemic model for information diffusion in MANETs*, Proceedings of the 5th ACM international workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems, 2002, pp. 54–60.

[62] Minseok Kwon and Sonia Fahmy, *Topology-aware overlay networks for group communication*, Proceedings of NOSSDAV 2002, 2002.

[63] T. Lee, M. Chams, R. Nado, S. Madnick, and M. Siegel, *Information Integration with Attribution Support for Corporate Profiles*, Proceedings of the ACM Conference on Information and Knowlege Management, 1999.

[64] Tobin J. Lehman, Stephen W. McLaughry, and Peter Wycko, *T spaces: The next wave*, Proceedings of the 32 nd Hawaii International Conference on System Sciences, 1999.

[65] M. Lenzerini, *Data integration: A theoretical perspective*, Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS 2002), Madison, Wisconsin, USA, 2002.

[66] M. Altinel and M.J. Franklin, *Efficient Filtering of XML Documents for Selective Dissemination of Information*, Proceedings of the 26th International Conference on Very Large Databases, September 2000.

[67] C. Marchetti, M. Mecella, M. Scannapieco, and A. Virgillito, *Enabling Data Quality Notification in Cooperative Information Systems through a Web-service based Architecture (Short Paper)*, Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003)(To Appear), Roma, Italy, 2003.

[68] C. Marchetti, M. Mecella, M. Scannapieco, A. Virgillito, and R. Baldoni, *Data quality notification in cooperative information systems*, Proceedings of the Workshop on Data Quality in Cooperative Information Systems, Siena, Italy, January 10-11, 2003, Proceedings of the International Workshop on Data Quality in Cooperative Information Systems.

[69] M. Mecella, M. Scannapieco, A. Virgillito, R. Baldoni, T. Catarci, and C.Batini, *The DaQuinCIS broker: Querying data and their quality in cooperative information systems*, Journal of Data Semantics (to appear) (2003).

[70] ______, *Managing data quality in cooperative information systems*, Proceedings of the 10th International Conference on Cooperative Information Systems, Irvine, CA, USA, 2002.

[71] Rene Meier and Vinnie Cahill, *Taxonomy of distributed event-based programming systems*, Proceedings of the International Workshop on Distributed Event Based Systems (DEBS '02), 2002.

[72] Sun Microsystems, *Java remote method invocation - distributed computing for java (white paper)*, 1999.

[73] ______, *Java API for XML-based RPC*, http://java.sun.com/xml/jaxrpc/, 2002.

[74] ______, *Java 2 platform, enterprise edition. version 1.4*, http://java.sun.com/j2ee, 2003.

[75] G. Muhl, *Generic Constraints for Content-Based Publish/Subscribe*, Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS), 2001.

[76] ______, *Large-Scale Content-Based Publish/Subscribe Systems*, Phd thesis, Technical University of Darmstadt, 2002.

[77] B. Oki, M. Pfluegel, A. Siegel, and D. Skeen, *The information bus - an architecture for extensive distributed systems*, Proceedings of the 1993 ACM Symposium on Operating Systems Principles, December 1993.

[78] L. Opyrchal, M. Astley, J. S. Auerbach, G. Banavar, R. E. Strom, and D. C. Sturman, *Exploiting IP multicast in content-based publish-subscribe systems*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000), 2000, pp. 185–207.

[79] M. P. Papazoglou and D. Georgakopoulos (eds.), *Service-oriented computing*, vol. 46, Communications of the ACM, no. 10, New York, NY, USA, ACM Press, October 2003.

[80] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha, *Efficient matching for web-based publish/subscribe systems*, LNCS, vol. 1901, Springer-Verlag.

[81] R. Piantoni and C. Stancescu, *Implementing the Swiss Exchange Trading System*, Proceedings of The TwentySeventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97), June 1997, pp. 309–313.

[82] G. P. Picco, G. Cugola, and A. L. Murphy, *Efficient content-based event dispatching in the presence of topological reconfiguration*, 23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA, 2003, pp. 234–243.

[83] P. Pietzuch and J. Bacon, *Hermes: a distributed event-based middleware architecture*, Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'02), 2003.

[84] ______, *Peer-to-peer overlay broker networks in an event-based middleware*, Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03), 2003.

[85] David R. Platt, *Understanding COM+*, Microsoft Press, 2001.

[86] D. Powell, *Group communication*, Communications of the ACM **39** (1996), no. 4, 50–97.

[87] R. Preotiuc-Pietro, J. Pereira, F. Llirbat, F. Fabret, K. Ross, and D. Shasha, *Publish/subscribe on the web at extreme speed*, Proc. of ACM SIGMOD Conf. on Management of Data (Cairo, Egypt), 2000.

[88] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, *Topologically-aware overlay construction and server selection*, Proceedings of IEEE INFOCOM '02, 2002.

[89] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker, *Application-level multicast using content-addressable networks*, Lecture Notes in Computer Science **2233** (2001), 14–34.

[90] T.C. Redman, *Data Quality for the Information Age*, Artech House, 1996.

[91] A. Riabov, Z. Liu, J.L. Wolf, P.S. Yu, and L. Zhang, *Clustering algorithms for content-based publication-subscription systems*, Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, 2002.

[92] ______, *New algorithms for content-based publication-subscription systems*, Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03), 2003, pp. 678–686.

[93] W. Rjaibi, K. Dittrich, and D. Jaepel, *Event Matching in Symmetric Subscription Systems*, Proceedings of the 12th Annual IBM Centers for Advanced Studies Conference (CASCON '02), 2002.

[94] A. Rowston, A. Kermarrec, M. Castro, and P. Druschel, *Scribe: The design of a large-scale notification infrastructure*, 3rd International Workshop on Networked Group Communication (NGC2001), 2001.

[95] A. Rowstron and P. Druschel, *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*, Proceedings of International Conference on Distributed Systems Platforms (Middleware), 2001.

[96] L. De Santis, M. Scannapieco, and T. Catarci, *Trusting data quality in cooperative information systems*, Proceedings of the 11th International Conference on Cooperative Information Systems (COOPIS 2004) (To Appear), 2003.

[97] K. Sattler, S. Conrad, and G. Saake, *Interactive example-driven integration and reconciliation for accessing database integration*, Information Systems **28** (2003).

[98] M. Scannapieco, A. Virgillito, C. Marchetti, M. Mecella, and R. Baldoni, *The daquincis architecture: a platform for exchanging and improving data quality in cooperative information systems*, Information Systems (to appear), Elsevier Science, 2003.

[99] B. Segall and D. Arnold, *Elvin Has Left the Building: A Publish /Subscribe Notification Service with Quenching*, Proc. of the 1997 Australian UNIX and Open Systems Users Group Conference, 1997.

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

[101] Yunxi Sherlia Shi, *Design of overlay networks for internet multicast*, D.Sc. thesis, Washington University, 2002.

[102] SIENA Web Site, *http://www.cs.colorado.edu/users/carzanig/siena/*.

[103] Dale Skeen, *An Information Bus Architecture for Large-Scale, Decision-Support Environments*, Unix Conference Proceedings, 1992, pp. 183–195.

[104] I. Stoica, D. Adkins, S. Ratnasamy, S. Shenker, S. Surana, and S. Zhuang, *Internet indirection infrastructure*, First International Workshop on Peer-to-Peer Systems, 2002.

[105] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, Proceedings of ACM SIGCOMM, 2001.

[106] ———, *Chord: A scalable peer-to-peer lookup protocol for internet applications*, IEEE/ACM Transactions on Networking **11** (February 2003), no. 1.

[107] A. Tanenbaum and S. van Steen, *Distributed Systems: Principles and Paradigms*, Prentice-Hall, 2002.

[108] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, *A peer-to-peer approach to content-based publish/subscribe*, Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03), 2003.

[109] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels, *Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining*, ACM Transactions on Computing Systems **21** (2003), no. 3.

[110] Y. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang., *Subscription Partitioning and Routing in Content-based Publish/Subscribe Networks*, 16th International Symposium on DIStributed Computing (DISC'02), October 2002.

[111] B. Whetten, T . Montgomery, and S. Kaplan, *A High Performance Totally Ordered Multicast Protocol*, Theory and Practice in Distributed Systems (1995), no. LNCS 938, 33–54.

[112] W.E. Winkler, *Methods for evaluating and creating data quality*, Information Systems (to appear) (2003).

[113] Zhichen Xu, Chunqiang Tang, and Zheng Zhang, *Building topology-aware overlays using global soft-state*, Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03), 2003.

[114] L.L. Yan and M.T. Ozsu, *Conflict Tolerant Queries in AURORA*, Proceedings of the Fourth International Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, Scotland, UK, 1999.

[115] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee, *How to model an internetwork*, IEEE Infocom, vol. 2, 1996, pp. 594–602.

[116] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, *Tapestry: A Resilient Global-scale Overlay for Service Deployment*, IEEE Journal on Selected Areas in Communications (2003).

[117] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz, *Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination*, 11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video, 2001.