UNIVERSITÀ DI PISA

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Dipartimento di Ingegneria dell'informazione

INCREASING RESILIENCE OF A PUBLISH-SUBSCRIBE SYSTEM

Relatori

Prof. Gianluca Dini

Ing. Alessio Vecchio

Candidato

Angelica Lo Duca

Anno Accademico 2006-2007

To my wonderful family.

Abstract

Existing publish-subscribe systems focus on functionality and routing and does not provide any guarantee in terms of security and fault tolerance. This work extends a particular publish-subscribe system, REDS (REcongurable Dispatching System), adding specific mechanisms to provide resilience. Security is provided through the concepts of secure path and access control. Fault tolerance is guaranteed increasing the number of links connecting two neighbors. Using these concepts, we illustrate the new architecture of REDS, called SEC-REDS, in which we have added new components, like the Security Manager and the Backup Connections Manager.

Acknowledgments

I would like to thank my advisor Prof. Gianluca Dini for trusting, encouraging, and supporting me all throughout my thesis. With advices and lively discussions, he has contributed a lot to the development of this work.
I am deeply grateful to Ing. Alessio Vecchio for his support and guidance especially in the past year.
I also would to thank Prof. Gianpaolo Cugola of Politecnico of Milano and Prof. Gianpietro Picco of University of Trento, because they have allowed us to study and extend REDS.
Last, but not the least, I would like to express my gratitude to Andrea Vigna, who has supported and encouraged me during difficulties.

Grazie di cuore a tutti!

Pisa, december 2007

Contents

Chapter 1

Publish-Subscribe systems

> Ma sendo l'intento mio scrivere cosa utile a chi la intende, mi è parso più conveniente andare drieto alla verità effettuale della cosa che alla immaginazione di essa.
>
> Niccolo' Machiavelli (1469 - 1527)
>
> Italian philosopher

1.1 Introduction

Publish-subscribe ([1], [2], [3], [4]) (here after called pub-sub) is a communication paradigm that supports dynamic, many to many communications in a distributed environment. It provides an asynchronous mechanism to exchange messages between *publishers* (senders) and *subscribers* (receivers). Subscribers express their interest in one or many classes of messages and they receive only that belong to these classes, whitout knowing any thing

about the senders originated them. This decoupling between publishers and subscribers allows a great scalability and a more dynamic network topology.

Publishing a message is said *event*, while delivering it *event notification*. This automatic delivery of messages is achieved through a third entity, called *Event Notication Service* or *Dispatching Network*, which receives events producted by the publishers and sends them to subscribers. The Event Notification Service (ENS, for short) represents the *middleware* and has to provide all the functions publishers and subscribers can use to subscribe, publish or remove their messages. For instance, instead of requiring publishers to identify destination addresses for their messages (potentially requiring multiple messages to multiple destinations), an ENS network can handle message routing in a way that avoids unnecessary message replications.

This event-based mechanism allows a decoupling between publishers and subscribers in the sense of:

- **time decoupling**: it is not necessary that both publishers and subscribers are active at the same time;
- **space decoupling**: publishers do not know neither subscribers identity nor their number, and vice versa.
- **synchronization decoupling**: publishers can produce events without waiting that someone receives them. Events are stored by the Event Notification Service, which notifies them to subscribers, while they are doing other operations.

Figure 1.1 shows a simple pub-sub system where subscribers S_1, S_2 and S_3 send subscriptions f_1, f_2 and f_3 respectively to some hosts in the ENS

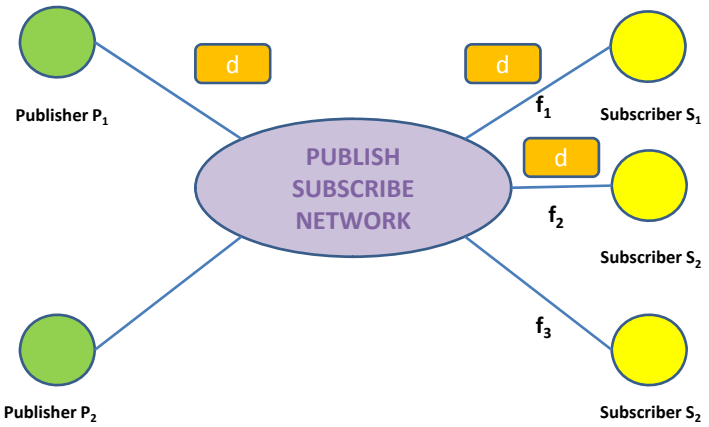

Figure 1.1: A publish-subscribe system

network. Publisher P_1 sends a datagram d to some entry point of the network. Message d matches f_1 and f_2 but not f_3 thus ENS sends d to S_1 and S_2. Note that the publisher only sends d into the network once and need know anything about the subscribers or subscription functions.

The earliest publish-subscribe systems used *subject-based subscription* ([5]). In such systems, every message is labeled by the publisher as belonging to one of a fixed set of subjects (also known as groups, channels, or topics). Subscribers subscribe to all the messages within a particular subject or set of subjects. A significant restriction with subject-based publish-subscribe is that the selectivity of subscriptions is limited to the predefined subjects. An emerging alternative to subject-based systems is *content-based subscription systems* ([6], [7], [8]). These systems support an event schema defining the type of information contained in each event (message). With content-based subscription, subscribers have the added flexibility of choosing filtering

criteria along multiple dimensions, without requiring definition of subjects.

1.2 Pub-sub systems design space

Existing publish-subscribe middleware differs along several dimensions ([9]):

- **the format of messages:** Messages can be untyped sequences of values like tuples, or untyped, record-like, messages or sequences of fields, each with a name and a value or, finally, typed objects;
- **the expressivity of the subscription language (filters):** The expressivity of the subscription language allows one to distinguish between *Subject-based*, in which the set of subjects is determined a priori (e.g. give me all the temperature events), and *Content-based*, in which subscriptions contain expressions (message filters) that allow clients to filter messages based on their content (e.g.: give me only the temperature events greater than ten grades);
- **the architecture of the dispatcher:** it can be *centralized* or *distributed*. In the first case, a single component is in charge of collecting subscriptions and forward messages to subscribers, while in the latter one, a set of message brokers organized in an overlay network cooperate to collect subscriptions and route messages. The topology of the overlay network and the routing strategy adopted may vary, but typically, a tree overlay and subscription forwarding are used;

- **the routing strategy (in presence of a distributed dispatcher):** it can be based on *message forwarding* or *subscription forwarding on a tree*. In the first case, every broker stores only subscriptions coming from directly connected clients, messages are forwarded from broker to broker and delivered to clients only if they are subscribed. In the second case, every broker forwards subscriptions to the others, subscriptions are never sent twice over the same link and messages follow the routes laid by subscriptions;

- **the forwarding strategy:** when a subscriber subscribes to a particular message (called *filter*), the broker adds it into a table, the *subscription table*. When that broker receives a message, it compares it with all the filters contained in the subscription table to determine to which subscribers the message must be delivered. The efficiency of this process may vary, depending on the complexity of the subscription language, but also on the forwarding algorithm chosen, which greatly influences the overall performance of the system.

1.2.1 Content-based routing

Content-based routing (CBR, [10], [11]) differs from classical routing in that messages are addressed based on their content instead of their destination. In conventional systems, the sender explicitly species the intended message recipients using a unicast or multicast address. Instead, in CBR the sender simply injects the message in the network, which determines how to route it according to the nodes interests. They identify the relevant classes of

messages based on their content, e.g., using key-value pairs or regular expressions. Therefore, in CBR it is the receiver that determines message delivery, not the sender.

Although it enables multi-point communication, CBR is not simply multicast. In network-level (e.g., IP) or application-level (e.g., topic-based publish-subscribe) multicast, the address of the multicast groups (or topics) used by the application must be defined a priori and made globally known or available. Moreover, a client joined to a given group receives all the messages addressed to that group, and only those. If the messages are to be received from multiple groups, the client must join all of them. Indeed, messages are conceptually partitioned in classes, and the binding between a message and its class is established by the sender. In contrast, in CBR message consumers define their own message classes; these select only the desired messages, need not be known to other clients, and can be arbitrarily overlapping.

CBR fosters a form of implicit communication that breaks the coupling between senders and receivers. Senders no longer need to determine the address of communication parties. Similarly, receivers do not know who is the sender of a message, unless this information is somehow encoded in the message itself.

The sharp decoupling induced by this form of communication enables one to easily add, remove, or change brokers or clients at run-time with little impact on the overall architecture.

1.3 State of the art

1.3.1 SIENA

SIENA (*Scalable Internet Event Notification Architectures* [12], [13], [14], [15], [16]) is a research project which realizes a generic *scalable pub-sub event notification service*. It's based on the content based networking.

Given that the primary purpose of an event notification service is to support notification selection and delivery, the challenge SIENA deals with is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* in the delivery mechanism. Expressiveness refers to the ability of the Event Notification Service to provide a powerful data model with which to capture information about events, to express filters and patterns on notifications of interest, and to use that data model as the basis for optimizing notification delivery, while scalability is the ability of the ENS to accommodate any growth in the future, be it expected or not.

1.3.2 REDS

REDS (*REconfigurable Dispatching System* [17]) is a publish-subscribe middleware designed to tolerate dynamic reconfigurations of the dispatching infrastructure. Its highly modular design decouples the management of reconfiguration from the other issues, and in general empowers developers with a high degree of flexibility. In fact REDS provides a modular architecture whose components can be easily changed to adapt to different deployment scenarios.

It is organized as:

1. a collection of peers, called *clients*, which publish and subscribe messages,

2. *the dispatcher*, which is responsible for collecting subscriptions and forwarding messages from publishers to subscribers. The dispatcher is realized as a distributed set of brokers interconnected in a overlay network, cooperatively routing the messages and subscriptions issued by the clients connected to them.

REDS is a framework (in the object-oriented sense) of Java interfaces and classes, which define:

1. a *client API*, enabling access to the publish-subscribe services;

2. A *broker API*, enabling access to the components inside the broker.

To support dynamic reconfiguration of the dispatching network, REDS brokers are structured in two layers: *Overlay* and *Routing*, which are independent.

The overlay layer is in charge of managing the topology of the overlay dispatching network. It offers services to build the overlay network and rearrange it based on input from upper layers.

The routing layer implements the main routing process by registering with the overlay component to be notified when subscriptions, messages and replies arrive from neighbors.

Hereafter REDS architecture is considered and extended. Why do we have preferred REDS to other pub-sub systems?

- Many scientific publications exist about REDS, that is REDS is object of interest for many researchers.
- The software is publicly available.
- Its developers are Italian and one of them has been a project partner of our Department.

1.4 Security issues and requirements for Pub-sub systems

In a wide-area pub-sub network, the pub-sub service must handle information dissemination across distinct authoritative domains, heterogeneous platforms and a large, dynamic population of publishers and subscribers. Such an environment raises serious security and trust concerns. This includes *access control* to the pub-sub infrastructure (and the data it transports), as well as the need to establish mutual *trust* between producers and consumers of data.

The security requirements [18] [19] [20] for a pub-sub system can be divided into the requirements for a particular application involving publishers and subscribers, and the requirements for the pub-sub infra structure:

- The **application**, comprising the publishers and subscribers. Publishers and subscribers may not trust each other, and may not trust the pub-sub network.
- The **infrastructure**, consisting of the pub-sub network that provides

services to the application. The infrastructure may not trust publishers and subscribers. Components of the infrastructure may not necessarily trust each other.

For example, providing a mechanism that defines who has what access to what information is mostly an application-level concern. It requires a definition of identity, authorization and access control within the pub-sub infrastructure. In the meantime, controlling who is able to change the subscription database maintained by the pub-sub service and restricting channel utilization are infrastructure-level protection issues.

1.4.1 Generic issues

The most important security issues are:

1. **Authentication**: it establishes the identity of the originator of a message. End-to-end authentication can be implemented outside of the pub-sub domain. If a public key infrastructure exists independent of the pub-sub network, end-to-end authentication can be accomplished by having publishers sign messages using their private keys. The subscribers can then verify a publishers identity by verifying the digital signatures attached to the message. The signing and verification operations occur outside of the pub-sub domain, and they can be administered independently.

2. **Information integrity**: it establishes that a message is not modified by an unauthorized node (a client or a broker) during its delivery. The standard means to provide information integrity is by using digital

signatures. A digital signature, when signed on the message digest with the senders private key, provides that the message content has not been changed since it is signed, and the message indeed originated from the sender. The provision of digital signatures can be largely independent of the pub-sub infrastructure.

3. **Subscription integrity**: it establishes that a subscription is not modified by a broker, that is, if a client C subscribes to a message A, the broker cannot change this subscription, for example specifying that C subscribes to a message B. This is a traditional access-control issue that can be solved with traditional means providing proper authentication and rights management.

4. **Service integrity**: it guarantees that messages are delivered correctly to right clients. A malicious broker could insert bogus subscriptions and act as a bogus subscriber to neighboring brokers. Moreover, it can ignore the routing algorithm entirely and route messages to arbitrary destinations or drop them completely.

5. **User anonymity**: it specifies that the source of a message (a publisher or a subscriber) cannot be recognized through that message. A publisher has to preserve his anonymity, without the Event Notification Service or the subscribers can know who has originated that event. Also a subscriber has to preserve his anonymity.

1.4.2 Confidentiality

Pub-sub systems introduce three novel *confidentiality issues*:

- **Information confidentiality**. When information being published contains sensitive content, publishers and subscribers may wish to keep information secret from the pub-sub infrastructure. The requirement of confidentiality against the infrastructure is in a fundamental conflict with the pub-sub model. By definition, the pub-sub network routes information based on dynamic evaluations of information content against user subscriptions. Keeping the information private from the routing hosts may hinder such evaluations and hence routing.

- **Subscription confidentiality**. User subscriptions can reveal sensitive information about the user, in which case the subscriber may wish to keep the subscriptions private.

- **Publications confidentiality**. In some applications publications have to be kept secret from those who are not legitimate subscribers. Publications confidentiality can be considered independent of the pub-sub infrastructure. For example, the publisher can distribute a group key to the subscribers using some out-of-band channel and encrypt the information content with the key. This ensures that only the subscribers with the right key can read the message. The drawback of this scheme is obvious: setting up a group key a priori, in essence, transforms the communication model into a traditional multicast model, and therefore minimizes the benefits of publish and subscribe. Alternatively, publishers can trust the infrastructure to maintain publication confidentiality.

Publishers send messages into the pub-sub system, which ensures that only registered users receive them.

1.4.3 Availability

As in other communication systems, denial of service attacks remain as a significant risk for pub-sub systems. In fact, malicious publications and subscriptions can be used to overload the system. In the general case, denial of service attacks are impossible to prevent. However, certain measure can be taken to minimize the probability of a wide spread denial of service attack. For example, a technique based on a limited number of publications by every publisher can be adopted.

1.4.4 Trust

Trust in pub-sub systems cannot be associated with specific producers and consumers, because one of the most important purposes of these systems is decoupling the two parts. This imposes the question of how the mutual trust between publishers and subscribers can be established. The obvious approach is to delegate some of the aspects of trust interaction to the pub-sub service. For instance, access control and secured delivery can be added to the pub-sub infrastructure. Unfortunately, this often implies that the infrastructure as a whole is trusted, and this is not the best solution to the problem. Another approach could be to split up publishers and subscribers in *groups of trust*. In this case, publishers belonging to a group have to trust only with subscribers and publishers belonging to the same group and not with those belonging to

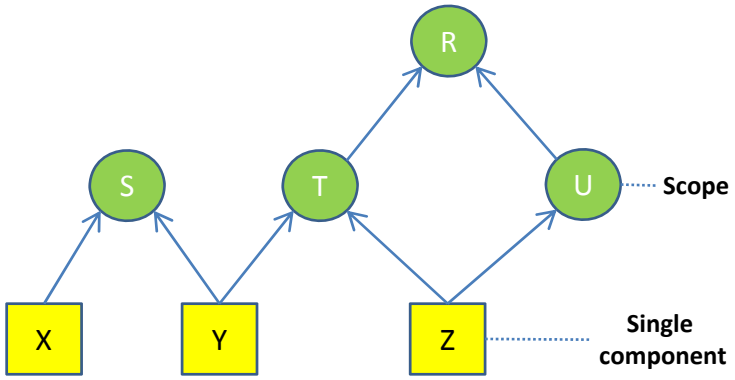

Figure 1.2: An scope graph.

the other groups. This is achieved through *scoping*.

Let consider the example of Figure 1.2, where R,S,T and U represent brokers, while X,Y and Z clients (in the figure a client is called *simple component*). The arrows represent the visibility, i.e. R sees T and U but not S, while T sees R and the clients Y and Z, and so on. Thus, a notification published by Z is delivered to Y and to any other consumers in T and U if their subscription matches. Also, it is visible in R if it matches the output interface of T or U, but it is not visible in S.

To guarantee that only clients belonging to a scope can produce and read messages from that scope, security services have to be implemented. These services base on the concept of *group*, described later.

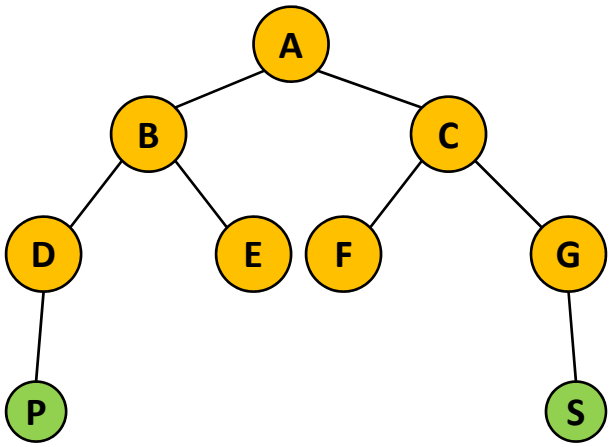

Figure 1.3: A standard ENS.

1.5 Fault tolerance in publish-subscribe systems

In general, a publish-subscribe system, as a generic distributed system, is not free of failures. In fact, hardware, software or all the dispatching network could crash. If we consider a simple ENS, it does not provide any guarantee in terms of *fault tolerance*. The *fault tolerance* is the capability of the system to guarantee that operations (like messages delivery) will continue even in the presence of faults.

Let consider the example of Figure 1.3. The ENS is made of the brokers A,B,C,D,E,F and G, the publisher P and the subscriber S. Let suppose that S subscribes to messages of the type published by P, so it will receive them. In normal conditions, the delivering is correct, and the messages go from P to S, passing through the path composed by D,B,A,C and G. But, if the link

between A and C drops, no path can be established from P to S, so S won't receive the messages. This situation brings to catastrophic consequences if P and S constitute a critic system. The previous example shows that publish-subscribe systems should maintain availability even at low levels of hardware/software/network reliability.

Fault tolerance, that should not involve users or system administrators, is achieved by:

- *data recovery*: it is the process of recovering data from damaged, failed, corrupted or inaccessible storage media. In order to provide data recovery, brokers should have a log file, in which they should store all the operations they make. This log file should be stored on a persistent storage media like a tape. Thus when a broker crashes, all its history can be recovered from the log file.

- *replication of brokers*: each broker could have a *twin* so all the information sent to it should be sent also to its twin. If it crashes, the twin could replace it. The mechanism could be extended so a broker could have many twins.

- *replication of links*: each broker is connected to its neighbors through many links, so if one of them drops, traffic goes on the others.

The rest of the chapter focuses on the third possibility, that is the *replication of links*.

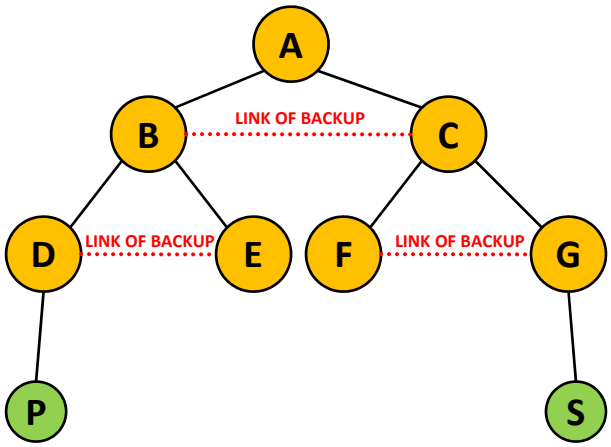

Figure 1.4: A tree with links of backup.

1.5.1 Replication of links in a tree ENS

Let suppose that the ENS is composed by brokers sorted to form a tree (see [21]), as the Figure 1.3 shows. If the link between two brokers drops, the connectivity of the tree is compromised. However a publish-subscribe system could offer a *dynamic reconfiguration of the network* (see [22], [23]), that allows the dispatching network to reconfigure dynamically if a broker departs from it. However, this mechanism does not resolve the problem, if a link does not drop, but it is overloaded. In this case, the dynamic reconfiguration of the network does not work, because no link drops.

To resolve the problem, that is to allow the dispatching system to work even in the presence of overload, a new architecture could be provided. In a standard tree every broker is connected to its sons and to its parent. The new architecture extends the connectivity, so a broker can be connected also to its brothers through a *link of backup*, as the Figure 1.4 shows. A broker

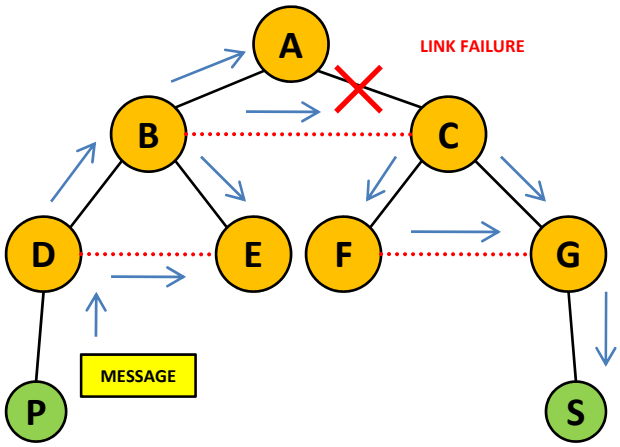

Figure 1.5: A link failure. Although the link between A and C drops, the message originated by P arrives to S.

sends messages that it receives both on the standard links (those connecting it to its sons and its parent) and on the links of backup. The receiver broker checks the received messages. If all those coming from the link of backup are also received from a standard link,it drops them, so it sends to its neighbors only the copy of the messages coming from the standard links. But, if there a message, that is not received from a standard link, as the Figure 1.5 shows, the broker can send to its neighbors that coming from the link of backup.

The link of backup has realized a form of *redundancy*.

1.5.2 A double link between brokers

The previous mechanism does not work when the topology of the ENS is not a tree. To resolve the problem, a new architecture can be provided.

As the Figure 1.6 suggests, two brokers can be connected through two connections, a *normal* one and a *backup* one. Normally, the traffic, wanting

Figure 1.6: A double link connection.

specific services (like security, a high throughput and so on) passes over the high performance link, while the standard traffic goes on the other link. If the low performance link drops, the service is not provided, because the traffic on it is not necessary, but if the high performance link drops, the traffic on it, is redirected over the link of low performance. However, if that traffic needs security guarantees, also the link of low performance must offer them. This can be achieved using standard techniques of confidentiality, integrity and so on.

Chapter 2

Infrastructure security

> Everything should made as simple as possible,but not simpler.
>
> Albert Einstein (1879 - 1955)
>
> German physicist

2.1 Introduction

In the previous chapter, two kinds of security have been presented: infrastructure and application. Hereafter infrastructure security is analyzed, that is security for brokers network.

In general, the dispatching network is composed of brokers. In the basic case (when there is no security), all these brokers trust each other. So a malicious broker can join the network with the same rights of the other brokers. Note that in this context the rights represent the capability of a broker to inject into the network publications and subscriptions. This means that the

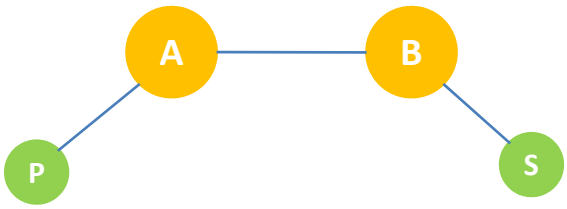

Figure 2.1: A dispatching network made of two brokers.

malicious broker can send to the others bogus publications/subscriptions. To understand the seriousness of the problem, let consider an example. Let suppose that the dispatching network is used as the infrastructure for a sensor network that monitors the temperature of a wood, and let suppose that the publishers are sensors of temperature, while there is only a subscriber, which is a computer that elaborates the received data, calculating the average temperature. If the temperature exceeds an established threshold, some counter measure is taken. If a malicious broker joins the network, it can inject bogus publications. So the subscriber receives those and could calculate a wrong value. This example shows that the problem does not concern only the brokers, but also the clients (a client is a publisher or a subscriber). In fact, there is no need for an adversary to build up a malicious broker to join the network and then inject wrong publications, because it can simply act as a publisher of false publications, as there is no control on clients.

The previous example shows the need of a secure architecture for the dispatching network, that:

- allows a broker to authenticate its neighbors (that is, brokers or clients directly connected with it),
- gives a subscriber the guarantee that it does not receive informations coming from bogus publishers, and a publisher the guarantee that its publications are not received also by malicious subscribers,
- gives a client the guarantee that the message delivery is secure, that is messages are not read or modified by an adversary during the rounting process.

In order to provide this mechanism, a generic publish-subscribe system model must be extended with the concepts of:

- *secure path*, and
- *access control*.

2.2 Secure Path

In general a broker A (or a client) wanting to connect to another broker B must know the URL of B. The broker A could establish a connection to B relying on a standard protocol, like TCP or UDP. However, A could have more complex needs, that is it could wish some guarantees in terms of security. In this case A could establish a connection using a secure protocol like SSL. In order to use the SSL protocol, A must present to B a *certificate*, that allows

it to authenticate itself to B. But also B must present a *certificate* to A, to authenticate itself. This mechanism relies on the *mutual authentication* of A and B.

The *certificate* sent from A to B and viceversa, is a standard certificate (like a *X509Certificate*), signed by a *Certification Authority*, that is not further specified in this context. It is made of all the informations used to identify a broker, like the subject, the organization and so on.

However, the authentication provided by SSL is not sufficient to authenticate a neighbor, in a publish-subscribe context. In fact, if the neighbor presents a certificate, a specific protocol like SSL does not check the subject of the certificate, it simply verifies whether it is valid or not. Instead, in a dispatching network a more complex test could be needed. Let suppose that a pub-sub system is used by a bank. An adversary could have a valid certificate, signed by a generic CA, so it could join the bank and receive all the information. This example shows that after having authenticated a neighbor, a second phase is needed, that is *authorization*. Each broker can store locally the list of only accepted certificates, that is an *access control list*. So although an adversary has a certificate, certainly it is not stored by the brokers of the bank, and it will not be accepted as trusted neighbor. However, if a broker does not contain a certificate in its local list, it may ask the dispatching network whether they have it. This procedure is described later.

Summarizing, a broker could wish to establish a connection with another broker, having some guarantees in terms of security. A such connection is called *secure connection*. In order to establish a secure connection, both the

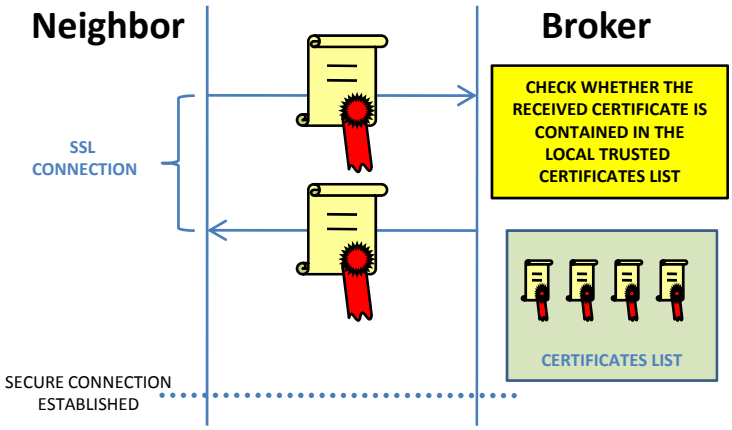

Figure 2.2: How a secure connection is established.

neighbors must pass two phases:

- *Authentication*, in which each neighbor provides to the other a *certificate*, signed by a generic CA. This certificate is analyzed by the receiver, which checks whether the certificate is valid or not, expired or not. If the certificate is valid and not expired, it is further analyzed in the second phase.

- *Authorization*, in which the broker checks whether the previous certificate is contained in the local *access control list* of certificates. If this phase is passed a secure connection is established between the two peers. However, if this phase is not passed a specific strategy can be adopted, specified by the *connection policy*, described later.

The Figure 2.2 shows what happens when a secure connection is opened. Firstly there is the negoziation of the link (e.g. SSL) parameters; then the *authentication phase* begins through the certificates. Each peer authenticates

itself sending its certificate to the other. If this phase is successful, each peer (only if it is a broker, a client does not further check the certificate) checks if its local access control list of certificates contains the received one. If the list of each neighbor contains the received certificate, the neighbor passes the *authorization phase*, and a secure connection is established between the two peers.

The concept of secure connection suggests the division of brokers and clients in *trusted* and *untrusted*. . Conceptually, a broker considers trusted another broker if it can assume that the informations received from that neighbor are not bogus. In practice, a broker (or client) is considered *trusted* by another broker if it is connected to it through a *secure connection* (like SSL), while it is considered *untrusted* if it is connected through a standard connection (like TCP or UDP).

Using the concept of trusted neighbor, we can give a new definition of *secure connection*. A connection is considered secure if all the messages passing on it cannot be read or modified by an unauthorized source, that is a node (a client or a broker) different from the two directly connected by that link. In our context, a secure connection is established between two brokers trusting each other. So a message passing on a secure connection cannot be read or modified by an unauthorized source. A chain of secure connections going from a publisher to a subscriber constitues a *secure path*. Let consider the example of Figure 2.3. The *secure links* between two neighbors are trusted, so a path exists from the subscriber to the publisher. This path is secure, because it is made by a chain of secure connections. In this case, the publisher has the guarantee that an untrusted third part cannot receive its

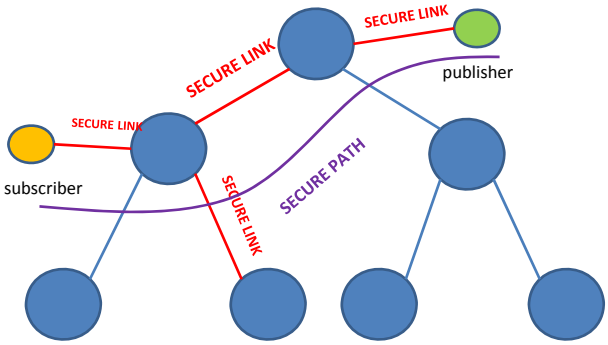

Figure 2.3: A secure-path example.

publications. At the same time, the subscriber knows that the publications it receives do not come from an untrusted publisher.

2.3 Group

In this paragraph we introduce the concept of *group*. Let consider the example of Figure 2.4. If the broker A trusts the broker B and the broker C, but not E, and if B and C trust D, and D trusts B and C but not F, a group made of A,B,C,D is built up.

Thus, a *group* is a collection of trusted brokers, in which every member of the group can reach the other members through a secure path. The concept of group is linked to that of *scoping*. Scoping implies that a broker is not able to see all the network, but it has a partial view of it. Let consider again the Figure 2.4. The broker E does not know that beyond A there are B,C and D, in the sense that if a publisher attached to E publishes a message,

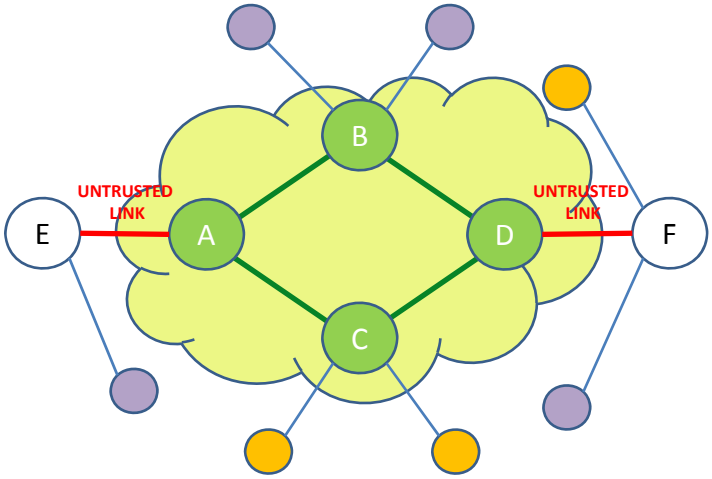

Figure 2.4: A group example: the cloud shows a group, made by A,B,C and D. The links beteween A and E, and D and F are untrusted.

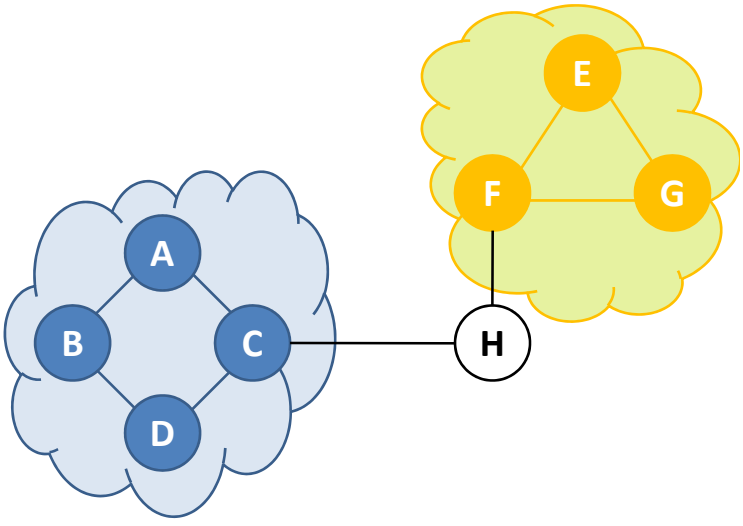

Figure 2.5: Isolated groups. A,B,C and D constitute a group and E,F and G form another group. The two groups communicate through the broker H, which is considered untrusted by both C and F so the two groups can't exchange secure informations.

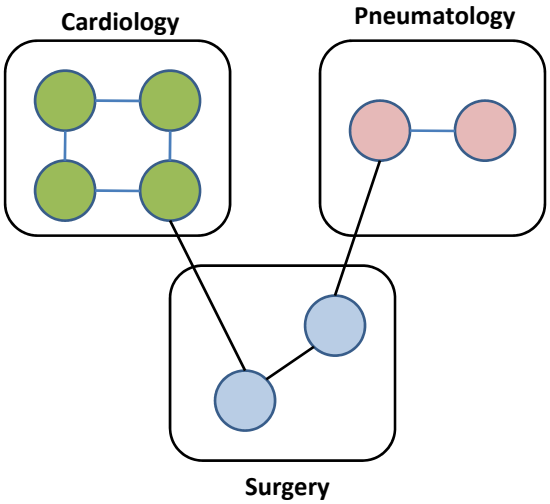

Figure 2.6: Hospital reparts. Conceptually the three reparts are independent each other. However, general informations, that is untrusted informations are delivered to all of them.

secure subscribers attached to A,B,C and D do not receive it. Note that a client is secure if it is connected to the broker with a secure connection. Only subscribers attached to F or unsecure ones attached to A,B,C or D can read the message. This suggests that many isolated groups may exist, as the Figure 2.5 shows.

The presence of many groups can be an advantage or a disadvantage, according to the points of view. Many separated groups are useful when the network is divided into *areas*, like reparts of an hospital, in which the general information must be received by all the subscribers, while particular ones must be received only by a repart rather than another. This is shown in Figure 2.6. However, the presence of many isolated group could constitute also a problem. A trusted subscriber belonging to a group will not receive informations coming from a publisher belonging to another group. This could

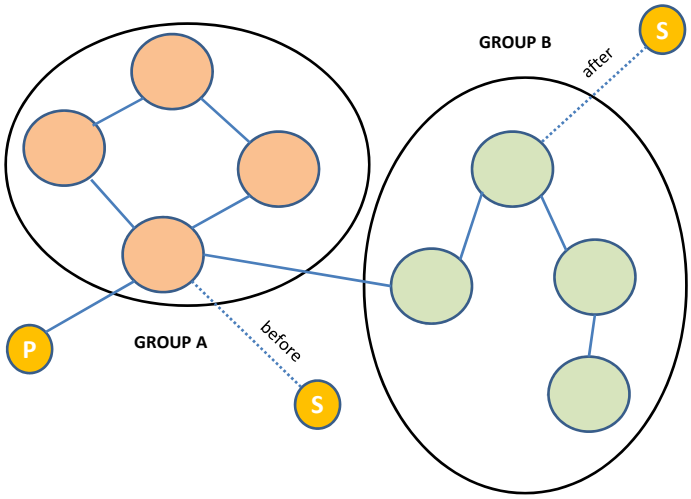

Figure 2.7: A mobile subscriber. The subscriber firstly attaches to group A, and secondly to group B.

seem right, but it is not. In fact, let consider the example of Figure 2.7. If a mobile subscriber attaches firstly to a broker belonging to the group A, it will receive informations from the publisher P, which belongs to the same group (here we are talking about secure clients), but if it latter moves and attaches to a broker belonging to the group B, it will not receive informations coming from P.

2.4 Certificate request

Let suppose that a client C wants to connect to a broker B through a secure connection. So it must present a Certificate. Let further suppose that the access control list of certificates of B does not contain that presented by C. In this case, B may simply refuse the connection request. However, it may ask the trusted brokers if someone in its local access control list of certificates

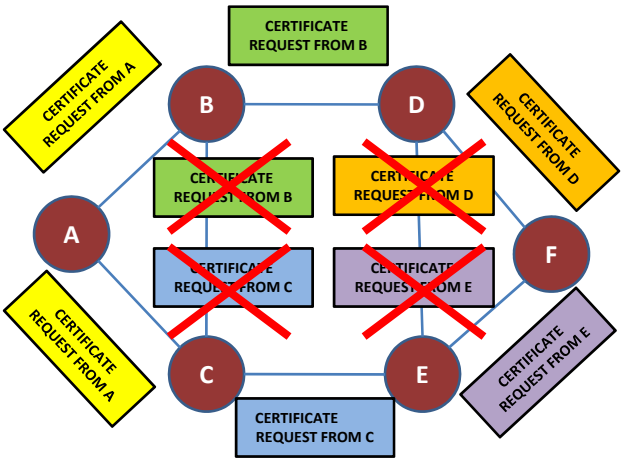

Figure 2.8: Certificate request flooding

contains that certificate. In this case B may accept the connection request of C. In order to provide this mechanism, when a broker receives a certificate that it does not know, it sends a `CERTIFICATE_REQUEST`, containing that certificate, to its trusted neighbors (only if they are brokers). When a neighbor receives a such message, it checks whether it contains that certificate in its local list. If the answer is positive, it builds a `CERTIFICATE_REPLY` to the sender, which can accept the certificate as trusted and so the connection request. However,if a neighbor does not know that certificate, it sends a `CERTIFICATE_REQUEST` to its trusted neighbors, except that from which it has received the request. In order to avoid loops, if a neighbor receives twice or many times the same request it drops it and does not send it to its neighbors. The Figure 2.8 shows a certificate request flooding.

The broker A sends the request to B and C. B sends it to C and D, but C drops it because it has already received the request from A. C sends the

request to B and E, but B drops it and so on until the request arrives to F. Let suppose that the broker F of the previous figure knows the certificate, so it sends a `CERTIFICATE_REPLY` to the neighbor from which it has received the message. In order to send the message to A, the reply must contain the chain of brokers from F to A. To explain this, let consider the example shown in the Figure 2.9, in which the brokers A,B,C and D are shown. When a broker sends a `CERTIFICATE_REQUEST` to its neighbors, it adds its *node id* to it (this operation is called *push*). So when the request arrives to the last broker, it contains the chain of brokers from A to D. A broker can establish to which neighbor it must send the `CERTIFICATE_REPLY`, making a *pop* in the stack of *node id* contained in the message. Obviously, also the `CERTIFICATE_REPLY` must contain that stack. If the source of the `CERTIFICATE_REQUEST` (i.e. A in the example) does not receive any answer in a time at least equal to the RTT of the network, it considers the certificate untrusted, so it can simply apply the specific connection policy, described later to the new neighbor (that is, the neighbor asking for the connection).

2.4.1 Managing untrusted connections

In the previous paragraphs, we have said that if a broker wants to open a secure connection towards another broker, it must pass both the *authentication test* and the *authorization test*. But, what happens if a neighbor does not pass at least one of those tests? In this case, the connection cannot be secure, so the neighbor cannot be considered trusted.

In order to manage those situations, each broker maintains a local policy,

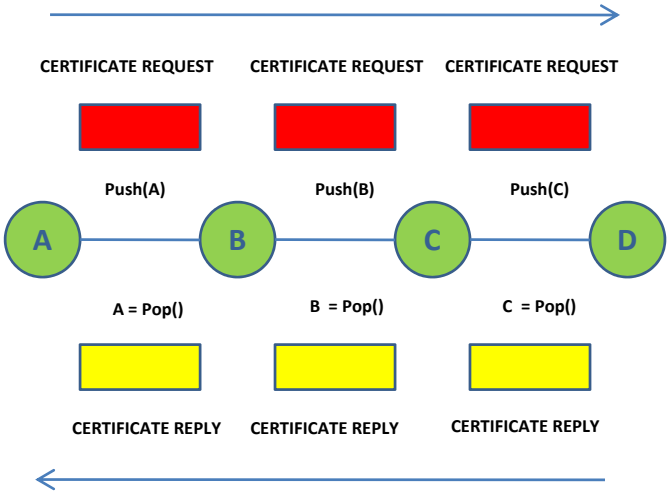

Figure 2.9: Certificate reply forwarding.

called *connection policy*, which establishes what must be done when a neighbor does not pass the *authentication test* or the *authorization test* or both of them. Many *connection policies* could be adopted. For example, the broker could simply refuse the connection (so the connection is closed) or it could accept that connection as untrusted.

2.5 Access Control

In the previous paragraphs, we have analyzed the concept of *secure path*, which has allowed us the division between *trusted* and *untrusted* neighbors. In this paragraph we consider trusted neighbors so the access control refers only to the trusted part of the dispatching network. In particular, messages coming from a trusted client (that is a trusted publisher or subscriber) can submitted to *access control*: each broker stores an *access control policy* so when it receives a message, it checks whether that message is consistent

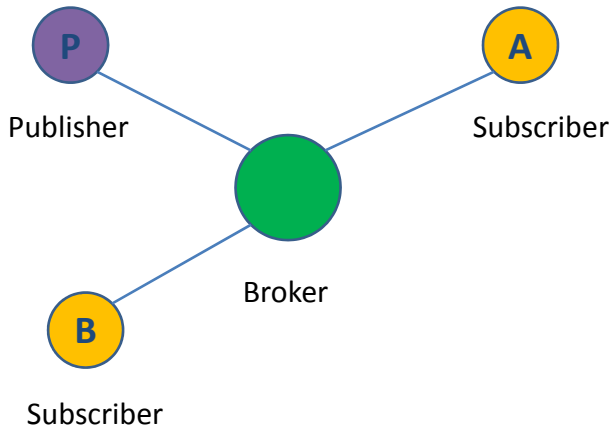

Figure 2.10: An example of bogus subscriber.

with its access rights. Only in this case, the client is allowed to publish or subscribe to that message.

Conceptually, a client wanting to connect to a broker must send it a tuple (S,A,R), in which it specifies that the subject S wants to perform the action A on the resource R. In particular, the subject is the same client, the action is publish or subscribe and the resource is the type of message. The broker compares the received tuple (S,A,R) with its local access control policy and if there is a match, the client is allowed to perform the asked operation on the asked resource.

However a problem arises. Let consider the example of Figure 2.10, where two subscribers and a publisher are showed. Let suppose that the local policy of the broker specifies that "only the subscriber A can receive the publications of P". So when A subscribes, it sends to the broker the tuple (A, subscribe, Messages of P) and then it will receive the messages. The subscriber B,

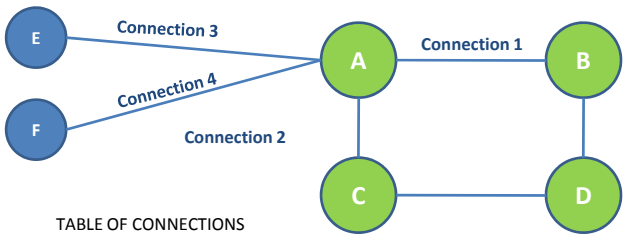

TABLE OF CONNECTIONS OF BROKER A

Connection	Node ID
1	B
2	C
3	E
4	F

TABLE OF ACCESS RIGHTS OF BROKER A

Node ID	Access Rigths
Subscriber E	(S_E ,R_E ,A_E)
Subscriber F	(S_F ,R_F ,A_F)

Figure 2.11: Broker A stores a *table of connections*, in which it stores for each connection the *node id* of the neighbor, and a *table of access rights*, in which for each client it stores its access rights.

also connected through a secure connection, subscribes sending the tuple (B, subscribe, Messages of P) so it will not receive any message. But if B is a malicious subscriber, it can send a bogus tuple (A, subscribe, Messages of P), as there is no control that the three elements of the tuple are true.

To resolve this problem, different strategies can be adopted. A simple solution is shown in the Figure 2.11: each node (that is a broker or a client) is characterized by a *node id*, that is unique in the whole network. For every connection each broker stores the *node id* of the neighbor so when it receives a message it is able to determine the *node id* of the sender of that message. Furthermore, the broker stores in a table for each *node id* which are the access rights, so when it receives a message, the broker determines the *node id* of the sender and checks if the tuple (S,R,A) contained in the message is substantial with the access rights stored in the table for that *node id*. If

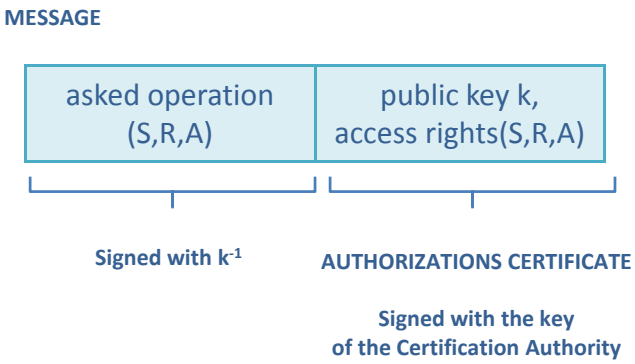

Figure 2.12: Message format.

the test is positive, the client is allowed to perform the asked operations. However, this solution supposes that the broker knows the access rights of all clients.

As alternative, we can introduce the concept of *Authorizations Certificate*. When a client wants to perform an action on a resource, it must present an *Authorizations Certificate*, signed by a Certification Authority not specified in this context. The Authorizations Certificate authorizes a subject to perform an action on a resource. Thus the Authorizations Certificate gives the client some particular access rights. In particular, the format of the message sent by the client is shown in the Figure 2.12. The message is composed of two parts, the Authorizations Certificate and the tuple (S,R,A), that specifies the asked operation. The Authorizations Certificate contains the public key of the sender and its access rights and it is signed by a generic CA. Optionally, it can also contain a content, described later. The tuple (S,R,A)

contained in the message is signed with the private key of the sender, whose corresponding public key is contained in the Authorizations Certificte. So a binding is created between the Authorizations Certificate and the asked operations. When a broker receives a such message, it controls that the Authorizations Certificate is valid, then it takes the public key from that certificate and uses it to decript the tuple (S,R,A) contained in the same message. Then the broker checks that the asked operations contained in the tuple are substantial with those specified in the Authorizations Certificate. If all these tests are passed, the broker accepts the subscription/publication contained in the message.

Summarizing, access control is achieved through two mechanisms:

- the local policy, stored by each broker, and
- the Authorizations Certificate, contained in the message sent by the client.

When a broker receives a message containing the Authorizations Certificate it calculates the *logic and* between the local policy and the policy specified in the Authorizations Certificate. The result gives the access rights of the client. In order to understand the mechanism, let consider the following example. Let suppose that the local policy of a broker is made of two rules:

- *allow all the clients to subscribe to a message of type XXXX*,
- *deny all publishers to publish messages of type YYYY.*

Let further suppose that a subscriber wants to send a subscription to the previous broker and its Authorizations Certificate is: *Allow this client to*

subscribe to messages of type XXXX. In this case, when the broker receives a such message, it checks the policy contained in the Authorizations Certificate and compares it with its local policy, making a *logic and*. In the example the broker calculates the *logic and* between *allow all the clients to subscribe to a message of type XXXX* (given by its local policy) and *Allow this client to subscribe to messages of type XXXX* (given by the Authorizations Certificate). The result, obviously, is *Allow this client to subscribe to messages of type XXXX*.

Furthermore, the mechanism of access control can be extendend. Infact, a publisher can send to the broker a message with new rules, in which it specifies which subscribers can receive that message. The broker adds these rules into its policy and floods them to its secure neighbors, that update their policy and flood them to their neighbors and so on. These rules are combined with the existing ones according to the specific *combining algorithm* specified by the local policy configuration file.

Let consider the example of Figure 2.13. Let suppose that the brokers of the network are all trusted. The publisher publishes a message with a new rule. The broker directly connected to it, receives the message and update its local policy. Then it sends an *message of update* to its secure neighbors (brokers). These brokers update their policy with the new rule and then forward the message to their secure neighbors, which finally update their policy. So the network is updated.

If a new trusted broker attaches to the dispatching network, the directly connected trusted broker floods to it all the new rules so it can update its policy. The introduction of this mechanism makes the system very flexible.

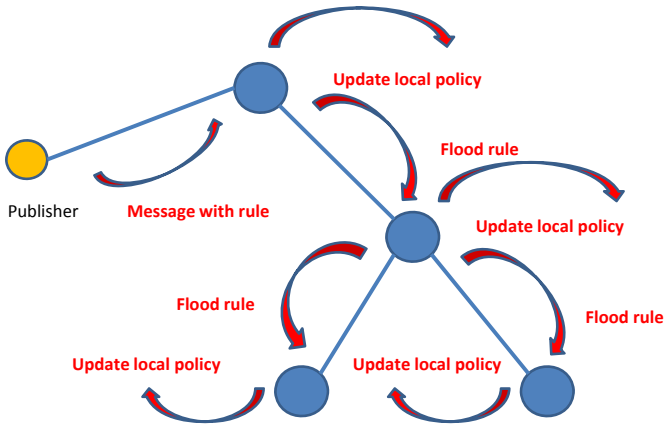

Figure 2.13: Rules flooding example

Let suppose that a subscriber tries to subscribe to a message of type A and let suppose that the dispatching network has no rule that allows that subscriber to subscribe. In this case, the subscriber request is moved into a temporary queue. When a publisher sends to the network a rule that allows all clients to access to the message of type A, that subscriber is removed from that temporary queue and is allowed to perform the asked operation.

2.5.1 Message content

As already said, optionally a message can have a content, that is not signed with the private key whose corresponding public key is contained in the Authorizations Certificate. In particular, only publications can have a content. The tuple (S,R,A) specifies the subject, the resource and the action corresponding to that publisher and in particular the resource represents only the heading of the publication. For example, the resource of a publication

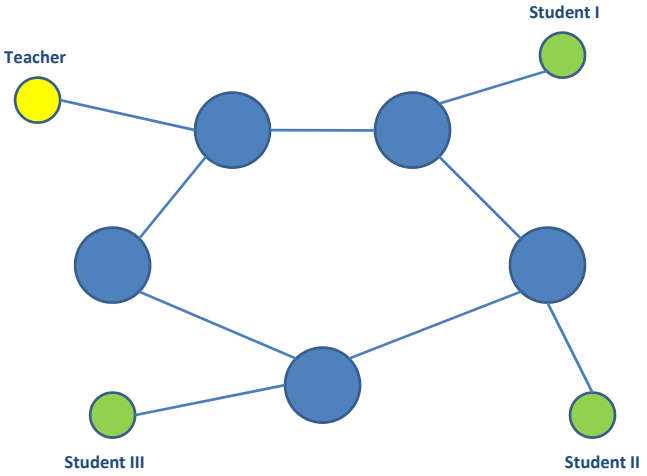

Figure 2.14: A teacher-students pub-sub system.

may be the *temperature*, and the content the effective value of it. So the resource contained in the tuple (S,R,A) is used only to identify the topic of the message, and the content represents the effective value of that topic. Here a problem arises. Given a set of subscribers with different access rights and needs and a publisher which publishes messages with content, we want to develop the following strategy: each subscriber receives only the part of that publication which it can receive, that is to which it has access rights. This mechanism corresponds to associate to a message content different *views*.

In general, the content of a message can be composed of many parts, and a publisher can specify which category of subscribers can read each part. Let consider the example of Figure 2.14. Let suppose that a teacher (publisher) publishes messages about phylosophy to his students (subscribers) and let suppose that students are divided in three categories (I, II and III). Let further suppose that students receive informations about phylosophy according

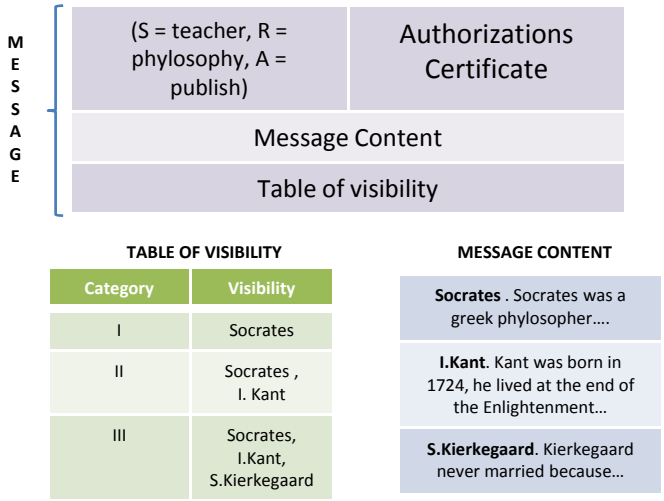

Figure 2.15: The message composed by the teacher.

to which category they belong. The teacher composes a message (shown in the Figure 2.15), in which the subject is himself, the action is publish and the resource (the topic of the message) is phylosophy. He also specifies the content of the message, and its visibility rules. Students belonging to the first category can see only information about *Socrates*, those belonging to the second category both information about *Socrates* and *Immanuel Kant*, those belonging to the third category all information. In order to provide this mechanism, the message contains a *table of visibility*, in which for each category, its visibility is specified.

When a broker receives a such message, it firstly analyzes if the tuple (S,R,A) matches that specified in the Authorizations Certificate. Then the broker delivers it to subscribers having access rights to that message (that is, subscribers allowed to read phylosophy messages). The Figure 2.16 shows the message delivering. Note that the last broker of the chain removes all

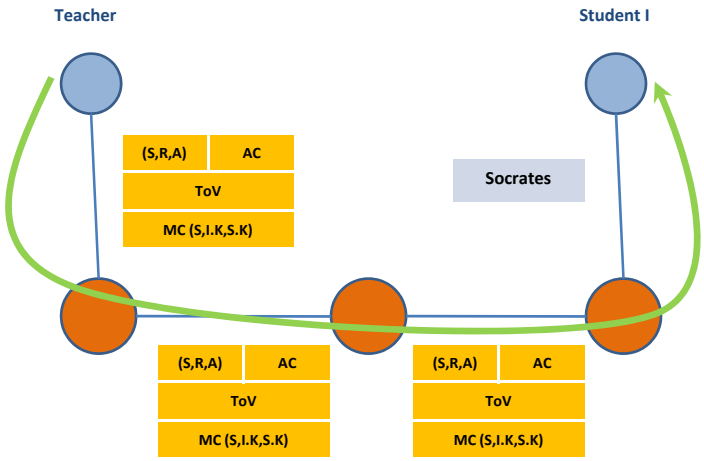

Figure 2.16: Message delivering. The student I (belonging to first category) will receive only information about Socrates.

the additional information (like the Authorizations Certificate, the table of visibility and the parts of the content for which the client has not access rights) carried out by the message, and sends to the client only those of interest.

2.6 REDS extensions

The standard REDS broker architecture does not provide any strategy to establish a secure path and access control. So we have extended this basic architecture to allow security guarantees.

In the new architecture of REDS, all the security aspects are managed by a new component, the *Security Manager*. Note that a *component* is a software object that is in charge of something and interacts with other components.

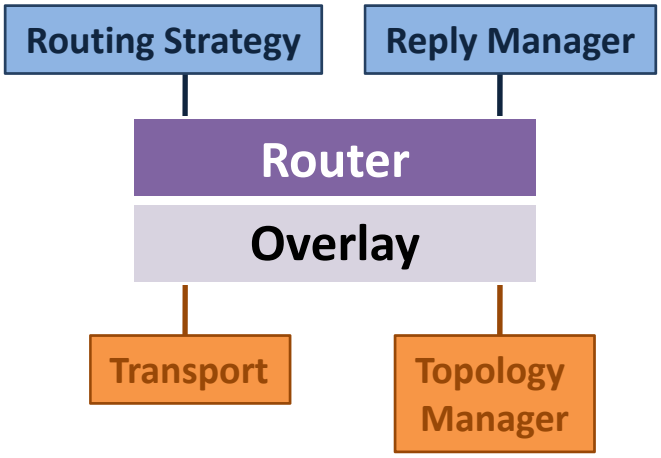

Figure 2.17: The standard REDS architecture. The Router is made of two sub-components, the Routing Strategy and the Reply Manager, while the Overlay is made of the Transport and the Topology Manager.

2.6.1 The Security Manager

The standard broker architecture is made by two components: the *Overlay*, and the *Router* (see Figure 2.17 for details). The former manages the topology of the dispatching network, while the latter provides a mechanism for the delivering of the messages. This basic architecture has been extended.

In the new architecture another component is added, the *Security Manager*, which is also the central component of this new architecture. It is in charge of security aspects, but in general it does not affect the Router and the Overlay components. Infact it is not a new layer in the REDS broker architecture, it is simply a new component so also in the new architecture the two old layers, the routing and the overlay, are mantained. The relationship among these components is shown in the Figure 2.18. The Security Man-

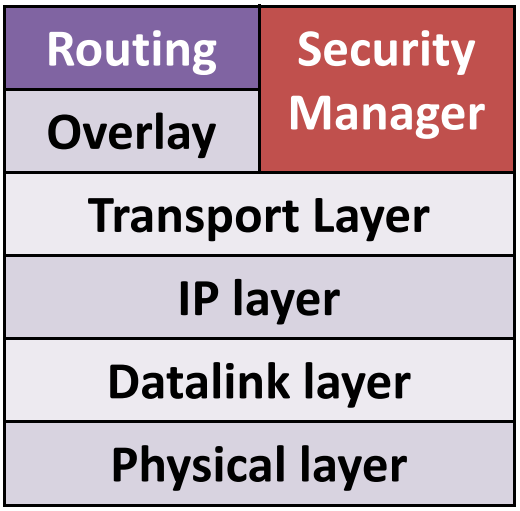

Figure 2.18: The OSI stack. The Security Manager, as the Routing and the Overlay, works at the application layer.

ager, as the other two components (i.e. the Router and the Overlay), works at the application layer of the OSI stack. It's consulted by the Overlay or by the Router, according to the specific operation. For example, the Transport, which is a sub-component of the Overlay, asks the Security Manager to check whether a neighbor can be accepted as trusted or not. This means that the Security Manager maintains the local policy for trustworthyness. Furthermore, the Routing Strategy, which is a sub-component of the Router, consults the Security Manager to know if a message can be delivered to a particular subscriber. This means that the Security Manager maintains also the local policy for access control.

Summarizing, when a broker B1 tries to open a new connection with a broker B2, it must pass two tests to become an effective neighbor of B2:

- the *topology test*, and

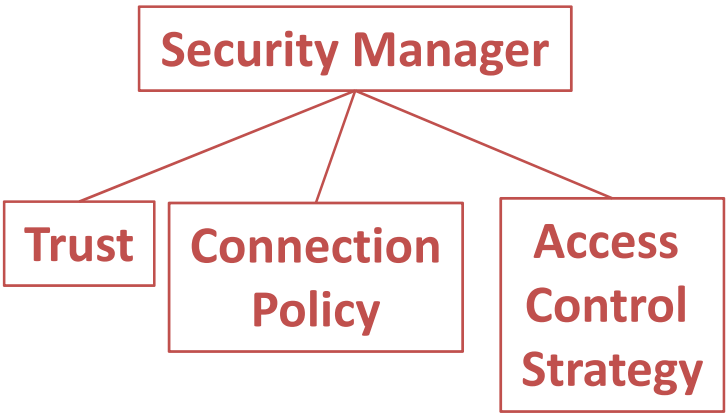

Figure 2.19: The Security Manager sub-components.

- the *security test*.

The *topology test* is mandatory and is made by the Topology Manager, a component provided by the basic REDS architecture. The *security test* is mandatory only if a neighbor wants to open a secure connection and establishes whether a neighbor can be considered trusted or not and in this last case what strategy must be applied.

As the Security Manager is in charge of many tasks, it is logic to divide it in many sub-components. Note that a sub-component is a software object that performs an action or many actions and interacts with other sub-components, but it cannot interact with a component. In practice, a component can be made of sub-components, but it is not necessary. As the Figure 2.19 shows, the Security Manager if made of the following components:

- the *Trust* and the *Connection Policy*, linked to the concept of secure path, and

- the *Access Control Strategy*, linked to the concept of access control.

2.6.2 Trust

The Trust component is in charge of maintening the access control list of certificates. In particular, it specifies the type of this access control list.

In the actual version of SEC-REDS two types of access control list of certificates have been implemented:

- the *Subject Based Trust*, in which the access control list of certificates stores only *subjects* of certificates. In this case, there is a match between the certificate received from the neighbor and the local access control list if there is a match between the subject of the received certificate and one of the subjects stored in the local access control list of certificates.

- the *Certificate Based Trust*, in which the access control list of certificates stores certificates. In this case, there is a match between the certificate received from the neighbor and the local access control list if there is a match between the received certificate and one of the certificates stored in the local access control list of certificates.

2.6.3 Connection Policy

The Connection Policy component, as its name suggests, is in charge of managing the *connection policy*, stored in the broker.

In the actual version of SEC-REDS two types of connection policy have been implemented:

- the *Brutal Close Connection Policy*, in which a connection is closed if at least one between the *authentication test* and the *authorizations test* is not passed.

- the *Untrusted Degrade Connection Policy*, in which a connection is degraded to untrusted if at least one between the *authentication test* and the *authorizations test* is not passed.

2.6.4 Access Control Strategy

To manage the access control, the Security Manager delegates to a new sub-component this task. The sub-component is the *Access Control Strategy*. In order to provide access control, the Access Control Strategy employs XACML, so the architecture of a broker has been extended with the elements to support XACML (e.g. the PDP), as the Figure 2.20 shows. In particular, the broker stores the policy into a configuration file (e.g. policy.xml). The XACML system receives as input a tuple composed by a *subject* S, an *action* A and a *resource* R. This tuple specifies that the subject S wants to perform the action A on the resource R. The XACML system asks the PDP (*Policy Decision Point*), if that subject can perform that action on that resource, and communicates the answer to the Access Control Strategy, which makes something, according to the local strategy.

In order to provide access control at the second level (that is, a message sent by a client contains its access rights through an *Authorizations Certificate*), a client can compose special messages, the *XACML Messages*which contain the *Authorizations Certificate*.

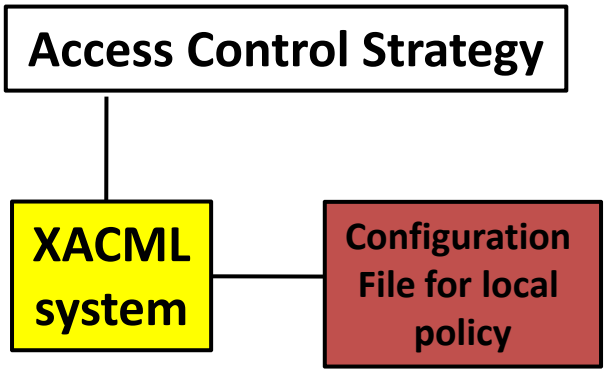

Figure 2.20: The Access Control Strategy.

2.7 Security issues provided by SEC-REDS

In the previous chapter a set of security issues for pub-sub systems has been analyzed. Here after we specify which of them are satisfied by SEC-REDS.

- *Authentication*. A client can authenticate the broker and viceversa to which it is attached through the certificate provided by it during the negotiation of parameters. Also two brokers directly connected can authenticate each other using a such certificate. But the authentication property is stronger; in fact, it allows a client to authenticate the identity of the originator of a message, that is for a subscriber to authenticate publishers. This can be achieved considering the concept of *secure path*. A secure path is made of secure connections, each of them has been established authenticating the neighbors. Thus, as the message comes from a secure path, a subscriber can consider the originator of the message (that is a publisher) as reliable.

- *Information integrity*. This property is again provided by concept of the secure connection, established between two peers. In fact, it guarantees that messages passing on it are not modified by an unauthorized source (i.e. SSL provides it). As a subscriber and a publisher are connected through a secure path and as a secure path is made of a chain of secure connections, that separately provide information integrity, we can conclude that this property is satisfied.

- *Service integrity*. Every broker is classified as trusted or not. An untrusted broker can't access to or insert trusted informations, so the service integrity is guaranteed.

- *Information confidentiality*. This property is again provided by the concept of the secure connection, established between two peers. In fact, it guarantees that messages passing on it are not read by an unauthorized source (i.e. SSL provides it). As a subscriber and a publisher are connected through a secure path and as a secure path is made of a chain of secure connections, that separately provide information integrity, we can conclude that this property is satisfied.

- *Subscription/Publication confidentiality*. If the publisher/subscriber is trusted, untrusted neighbors don't know that it exists, because neighter trusted subscribers receive untrusted informations nor trusted publications are delivered to untrusted subscribers.

- *User anonimity*. It is not provided.

- *Availability*. It is not provided. In fact, Denial of Service may occur.

Note that all these security aspects are been analyzed considering the dispatching network as trusted. If this hypotesis is not applicable, the previous analysis is not valid.

Chapter 3

Network redundancy

> Quicquid praecipies, esto breuis, ut cito dicta percipiant animi dociles teneantque fideles. Omne superuacuum pleno de pectore manat.
>
> Qualunque cosa tu ti proponga di insegnare sii breve, per modo che la mente apprenda subito senza fatica e ritenga fedelmente ciò che hai detto: tutto il superfluo trabocca dall'animo pieno.
>
> ---
>
> Quintus Horatii Flacci (65 a.C. - 8 a.C)
>
> Latin poet

3.1 Introduction

In the previous chapter we have analyzed security aspects, in terms of confidentiality, integrity and access control. In this chapter we will develop a strategy to build a robust pub-sub system, that is a tolerant failure system.

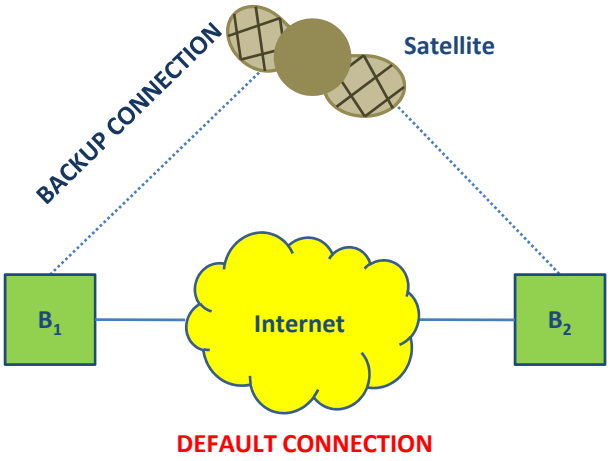

Figure 3.1: A backup connection example. The two brokers are connected with a standard connection (Internet) and a backup connection (through satellite).

A broker (or a client) may connect to another broker through many connections. In this case, one of them is considered of *default*, the others of *backup*, as shown in Figure 3.1.

Usually traffic is delivered using the default connection, but if something happens on it (e.g. interrupted, closed, overloaded connection) traffic goes on the first backup available connection. In order to provide this mechanism, many strategies can be adopted. A simple strategy consists in setting up many connections (all active), of which one is considered of default, the others of backup. This simply mechanism, however, wastes a lot of resources to mantain active many connections (e.g. in terms of bandwidth). In contrast this strategy is very simple, because when the default connection drops, one among the backup connections is elected as default. However this strategy is very simple, so a more complex mechanism must be developed.

CONFIGURATION FILE

Default Connection	Internet
Backup Connection	Satellite
Number of tentatives	4
Timeout	3 sec.

Figure 3.2: A backup configuration file (backup strategy).

In general a connection (default or backup) is established between two peers (neighbors), in particular one of them asks for the connection (i.e. tries to open the connection invoking the method `open()`) and the other listens to. The former is called *master*, the latter *slave*. This mechanism is implicit in REDS. Furthermore it is the master that establishes whether it must open a backup connection or not and specifies which actions the slave must perform in order to set up a such connection. In order to do it, the master must have a configuration file, that specifies which actions it must perform. Let consider again the Figure 3.1. Let suppose that B1 is the master and B2 the slave. Let further suppose that B1 has a configuration file (shown in Figure 3.2), in which it is specified that normally the two brokers are connected through the Internet connection, but if this drops, four tentatives are made to restore it. If all these tentatives fail, B1 tries to open a backup connection, using the satellite. Periodically, it checks whether the default

connection is restored. If the answer is affirmative, the backup connection is closed (as it is more expensive), and traffic goes on the default connection. As the example suggests, the two brokers have different importance, in fact the master decides what must be done and informs the slave of that.

3.2 SEC-REDS extensions

In order to provide fault tolerance, that is based on backup connections between two brokers, another component is added to the REDS architecture, that is the *Backup Connections Manager*, which works at the transport layer (as shown in Figure 3.3), manages all the connections of backup and notifies the transports if something happens (e.g. passing from a SSL connection to a TCP connection). Note that as the *Backup Connections Manager* works at the transport layer, the upper layers are not aware of the existence of backup connections. Furthermore they are not informed if traffic goes on a default connection or on a backup one.

The actual version of SEC-REDS provides two strategies to manage backup connections:

- the *Active Backup Connections*, and
- the *On Demand Backup Connections*.

3.2.1 The Active Backup Connections

In this simple strategy the *Backup Connections Manager* mantains for each neighbor the list of all the available connections. Among those, one

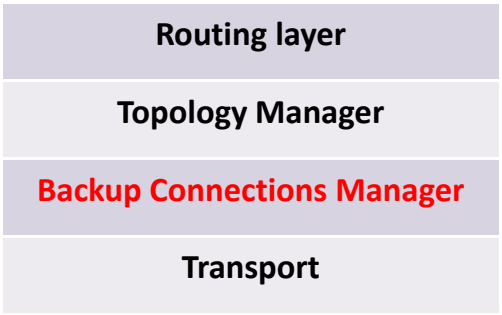

Figure 3.3: SEC-REDS architecture. Between the Transport and the Topology Manager a new component is added, the Backup Connections Manager.

is of default and the others are of backup, as shown if Figure 3.5. All the connections are active, in the sense that they waste resources, but only one (that of default) is used to deliver messages. In particular, each connection is identified by a *connection ID*, which must be unique for a neighbor, but in general it is unique in the whole ENS. So each neighbor (identified by a *neighbor id*) can set up many connections, each of them identified by a *connection id*. A *neighbor id* is unique in the whole network, but also a *connection id* is unique in the whole network.

A neighbor (master) can open a connection towards another neighbor (slave) invoking the method `openLink(String url)`, in which it specifies the *url* of the slave. The first connection opened is considered of default. If that method is invoked again on the same url, a backup connection is created. If that method is invoked many times, many backup connections are opened. The Figure 3.4 shows the effect of invoking many times the

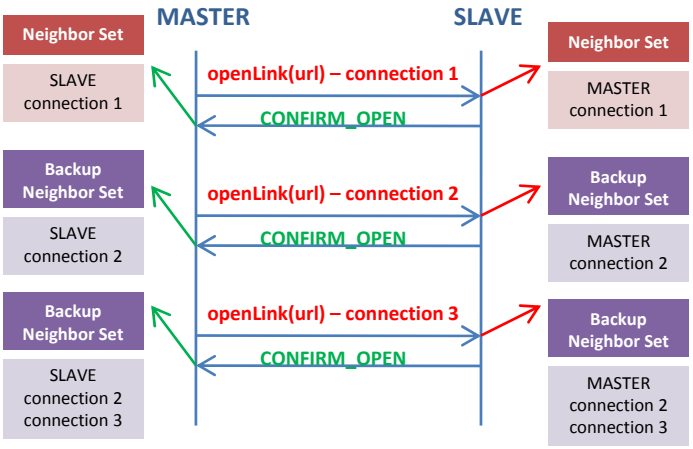

Figure 3.4: The effect of calling many times the method openLink(url) on the same url. After the first call, both the master and the slave add the neighbor in the neighbor set (which mantains the list of all neighbors). After the other calls, both the master and the slave add the neighbor in the list of backup neighbor set (which mantains the list of all backup connections).

method `openLink(String url)`. If the default connection drops, one of the backup connections (i.e. usually the first available) is elected new default connection, so traffic goes on it. If the old default connection is restored, it is added as backup connection.

A broker may wish to change its default connection towards another broker. In this case, the previous mechanism must be extended. Let consider the example shown in the Figure 3.6. The broker B1 is connected to B2 through three connections, A (default), B and C (backup). Let suppose that B1 wants to set the default connection to C. In order to perform a such action, it must inform B2 so also B2 can set the default connection to C. As shown in the Figure 3.7, B1 sends a messsage to B2, containing the request to change the default connection. Note that this message is delivered using

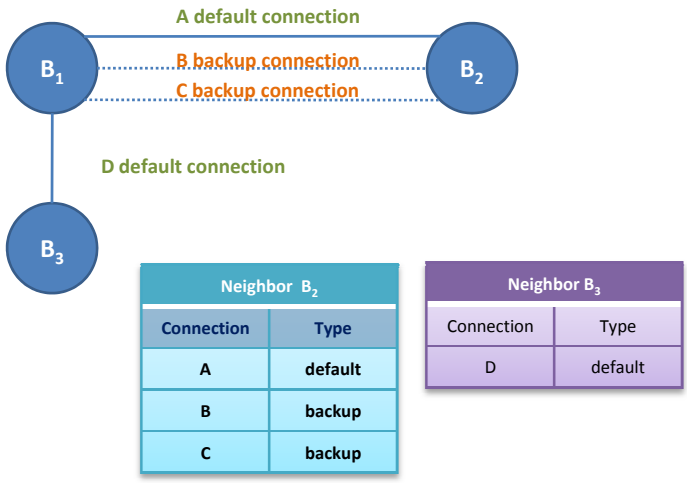

Figure 3.5: The tables show the lists of connections of broker B1. B1 is connected to B2 through the default connection A and the backup connections B and C. Furthermore, it is connected to B3 through the default connection D.

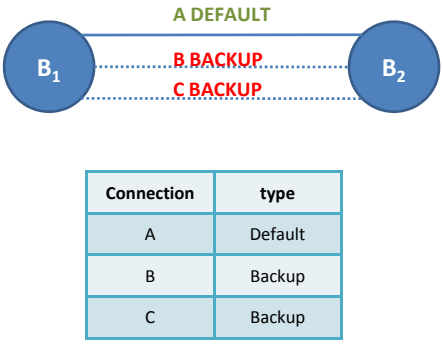

Figure 3.6: The two brokers are connected through three connections, A (default), B and C (backup).

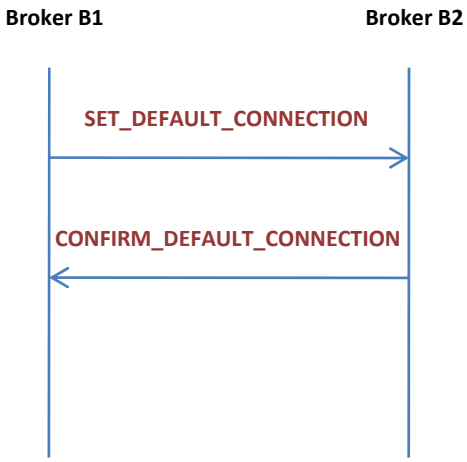

Figure 3.7: In order to change the default connection, B1 sends to B2 a message, set default connection, in which it specifies which connection must become the new default connection. B2 replies with a confirm default connection message.

the old default connection (A). When B2 receives a such message, it checks whether it can accept C as new default connection or not. If the answer if affirmative, it sends B1 another message (a confirm default connection). Eventually, C becomes the new default connection.

In order to set a default connection, a broker must invoke the method `setDefaultConnection(neighborID,connectionID)`, in which it specifies the *neighbor id* and the new default connection (identified by an id). The effect of this call is to change the default connection for the neighbor identified by the *neighborID* to that specified by the *connectionID*.

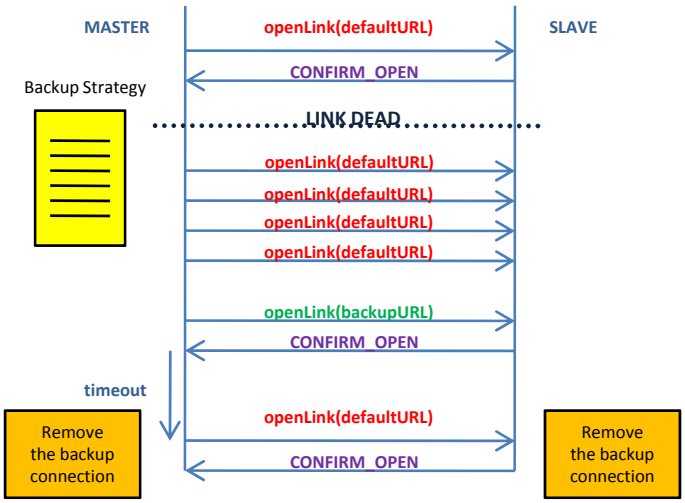

Figure 3.8: The On Demand Backup Connections mechanism.

3.2.2 The On Demand Backup Connections

The mechanism described in the previous paragraph is very simple. In particular, it does not work well when a system has not many resource. Let consider a sensor pub-sub system, in which each broker is a sensor. Let suppose that this system is used to measure glaciers temperature. Let further suppose that each sensor is put in a glacier. In general a sensor has limited resources (i.e. battery) so the previous strategy (many active connections, of which only one is of default) is not the best solution, as sensors cannot be removed from their place when their batteries die. Resources will have to last any more.

In order to satisfy a such need, another strategy is adopted, that is the *On Demand Backup Connections*. The previous mechanism (Active Backup Connections) is *weakly asymmetric*, as the master and the slave execute the same actions (however the master opens the connection). The new mecha-

nism (On Demand Backup Connections) is *strongly asymmetric*, as the master and the slave execute different actions. The master mantains a *backup strategy* (that is a configuration file, like that shown in the Figure 3.2), in which its backup policy is specified. Note that the slave has not a *backup strategy*. To explain the mechanisms, let consider the Figure 3.8. The master tries to open the default connection towards the slave, invoking the method `openLink(defaultUrl)`, the slave receives a such request and accepts it. As effect, the two neighbors are linked. If the connection drops, the slave simply notes it and closes the connection. In contrast, the master takes from the *backup strategy* what it must do. Let suppose that the *backup strategy* is that shown in the Figure 3.2. So it tries to restore the default connection. If this tentative is successful, the master does not execute other actions, but if it fails, the master retries for other three times. If all those tentatives fail, the master tries to open a backup connection (the satellite of the figure), invoking the method `openLink(backupUrl)`. Let suppose that this tentative is successful. In this case a new connection is established. Note that the slave is not aware of the strategy adopted by the master. It is also not aware that the new established connection is a backup connection. When the backup connection is established, the master starts a timer. When the timeout goes off, the master tries to restore the default connection. If this tentative fails, it starts a new timer, when the timeout goes off, it retries to restore the default connection and so on. If the tentative is successful, the master closes the backup connection. When the slave accepts a new connection from the same neighbor, it closes the old, as it was of backup, and sets up the new (the default). The effect is that master and slave are linked through the default

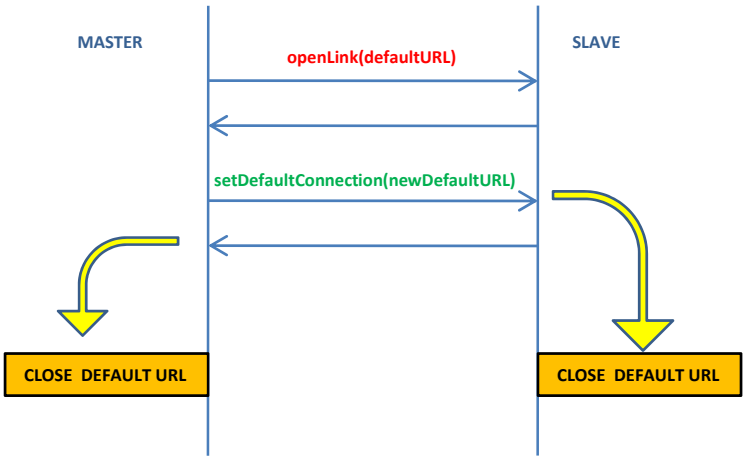

Figure 3.9: Changing a default connection using the On Demand Backup Connections mechanism.

connection.

As the *Active Backup Connections* mechanism, also the *On Demand Backup Connections* one allows a neighbor to change its default connection towards another neighbor. This can be achieved invoking the method `setDefaultConnection(newDefaultURL)`. The mechanism is showed in the Figure 3.9. Firstly the master opens a default connection towards the slave, invoking the method `openLink(defaultURL)`. A connection is established between the two peers. If the master wants to change its default connection, it invokes the method `setDefaultConnection(newDefaultURL)`, specifying the new default URL. As effect of this call, both the master and the slave close the old default connection, and they remain linked through the new default connection.

Chapter 4

SEC-REDS implementation

> Tutte le cose nel loro complesso sono migliori delle superiori prese da sole.
>
> SANT'AGOSTINO (354-430)
>
> Christian philosopher

4.1 Introduction

As said in the previous chapter the standard REDS architecture has been extendend. The standard REDS provides a generic implementation of the routing and overlay layers (called *GenericRouter* and *GenericOverlay*, respectively), which don't consider the security aspects. So, the first step towards a security architecture is to extend this basic service, with a *Secure Router* and a *Secure Overlay*. This is shown respectively in the Figures 4.3 and 4.4.

The pink components (the term *component* refers, as already said, to a software object that is in charge of something and interacts with other

components. It can be made of sub-components) shown in the figures are not provided with the basic REDS architecture and are been implemented in its extension. The figures don't show the sub-components of each layer (like the *TopologyManager* or the*RoutingStrategy*), but, obviously, also those are been extended in the new architecture to support security. The only component described in the next lines is the *Transport*, because it is more complex than the others.

The Transport components provided with REDS are the UDP and TCP. In the new architecture a *SecureTransport* is added, as shown in the Figure 4.5. A particular implementation of the SecureTransport is also provided, that is the *SSLTransport*, which gives the clients many security guarantees, like integrity and confidentiality of information. In order to provide both standard and secure connections, another component is added, the *ComplexTransport*, that manages them. So a secure broker must implement a *ComplexTranport* object in order to provide both standard and secure connections.

Also the client API has been extended to support security. In the specific case (as illustrated in the Figure 4.6), the *DispatchingService* interface has been extended with the *SSLDispatchingService*, which, as the name suggests, provides the SSL service.

If a client wants to open a secure connection, it must create a *SSLDispatchingService* object, and it must specify the url of the broker to which it wants to connect. In particular, it must present a *certificate*, that in the specific case is provided from command line (following the standard specifications of the JVM).

In a REDS environment, a subscriber sends to the ENS particular messages, called *filters*. A filter is a message sent by a subscriber that specifies to which type of messages it wants to subscribe. Thus clients are classified on the basis of their filters. The standard REDS architecture provides a generic interface, called *Filter*, that can be implemented according to the specific needs. REDS provides a *TextFilter* and a *PTreeFilter*. In the new architecture a new Filter is added, the *XACMLFilter*, that is checked by the Access Control Strategy of every broker. The interface Filter and its implementations is showed in the Figure 4.1. The XACMLFilter takes as input an *Authorization Certificate*, that is made of the tuple (subject,action,resource). This tuple is submitted to the XACML system of every broker through the Access Control Strategy.

In a REDS environment, a publisher sends to the ENS messages, each of them having a specific topic. The interface *Message* represents this concept, and the REDS architecture provides two implementations of it, the *TextMessage* and the *PTreeMessage*. Note the duality with the Filter implementations. In the new architecture, a new implementation is provided, the *XACMLMessage*, which is submitted to access control. The Message interface and its implementations are showed in the Figure 4.2. As the *XACMLFilter*, also the *XACMLMessage* takes as input an *Authorizations Certificate*, that contains the tuple (subject,action,resource), which determines who is the publisher, which action it wants to execute and which is the resource it wants to publish. Thus the resource is the topic of the message. Indeed, an XACMLMessage can have also a content, that can be specified using the method *setContent()*. If a publisher wants to establish a private access con-

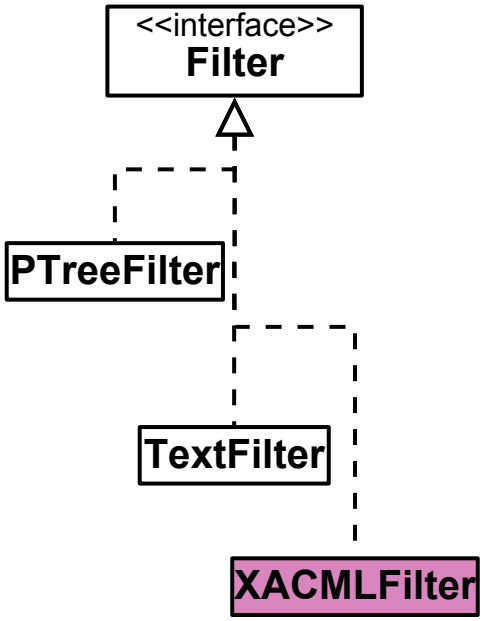

Figure 4.1: The **Filter**.

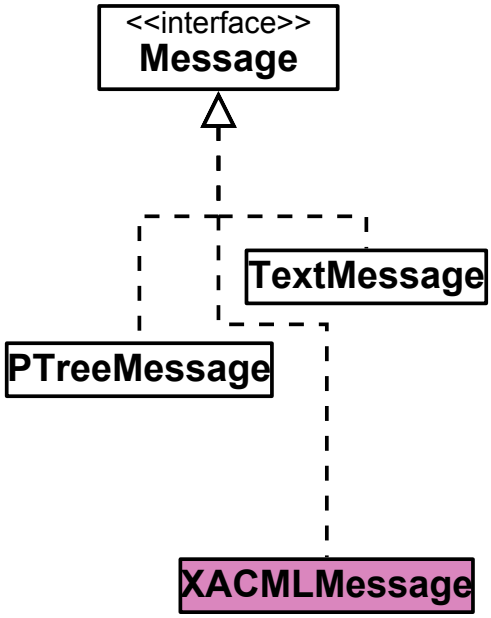

Figure 4.2: The **Message**.

trol policy on its message, it can invoke the method *setRule()*, that allows it to establish which subscribers can receive its messages.

4.2 The SecurityManager

As discussed in the last chapter, the *Security Manager* is the main component of the new architecture. The security extension of REDS provides a generic interface called *SecurityManager*, and a specific implementation, the *GenericSecurityManager*. This is shown in the Figure 4.7. In order to support the *security test*, two components are implemented: the *Trust* and the *ConnectionPolicy*. The former, as described previously, specifies whether

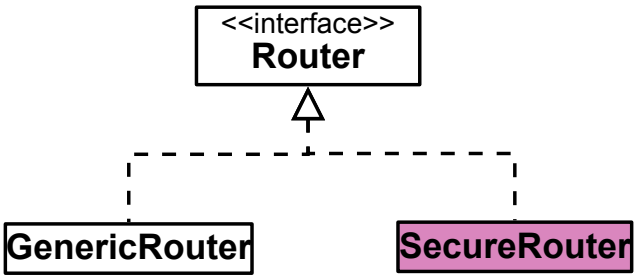

Figure 4.3: The **Router**. The Router interface is implemented by the GenericRouter (provided by REDS), which provides the basic services, and by the SecureRouter (extended by the new architecture), which provides security guarantees.

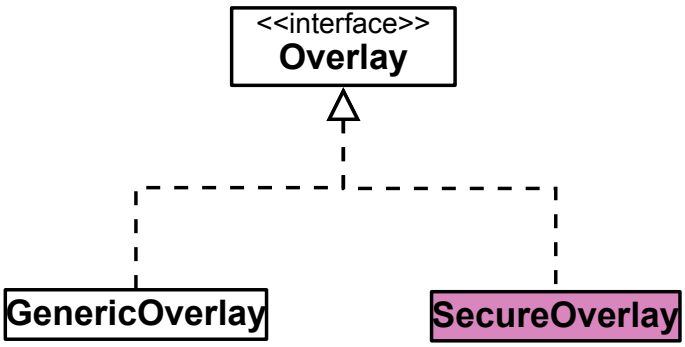

Figure 4.4: The **Overlay**. The Overlay interface is implemented by the GenericOverlay (provided by REDS), which provides the basic services, and by the SecureOverlay (extended by the new architecture), which provides security guarantees. The figure doesn't show the sub-components of the overlay layer, like the *TopologyManager*.

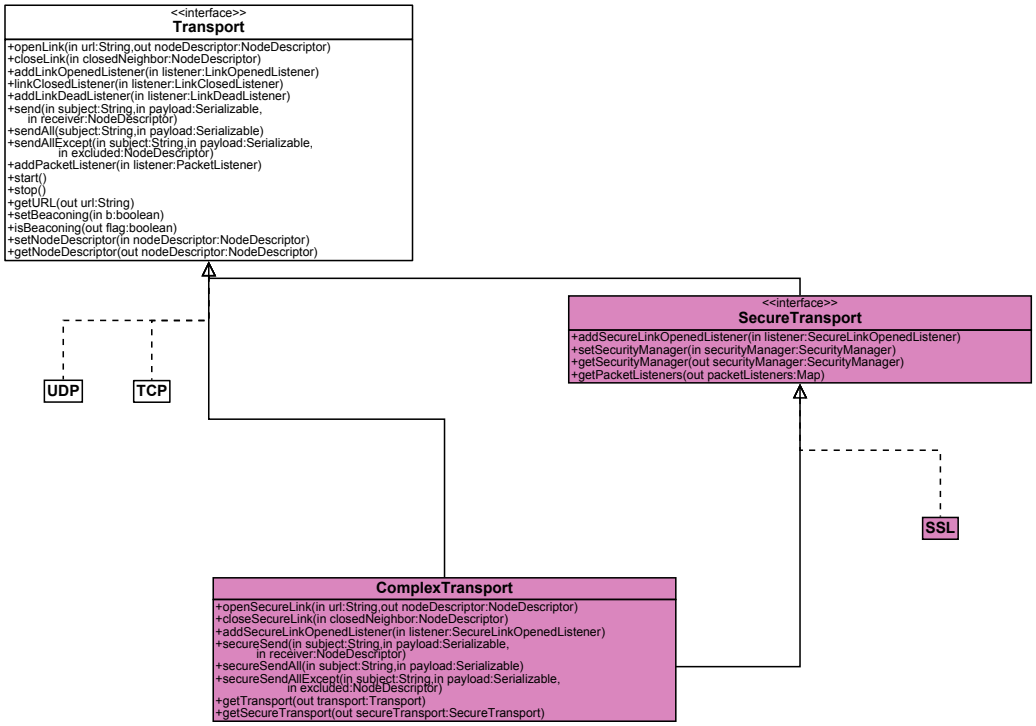

Figure 4.5: The **Transport**. To the standard Transports (UDP and TCP), the SSL transport is added. It's a SecureTransport. The last defines an interface that in the future could be extended by another type of protocol. The figure shows also The ComplexTransport component, which contains a Transport and a SecureTransport object. It's used to manage both standard and secure connections.

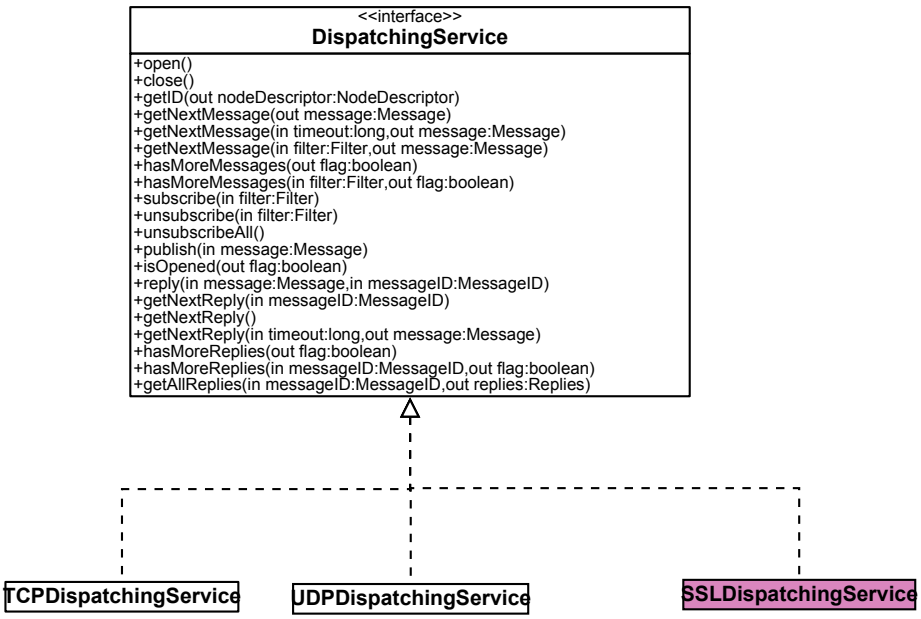

Figure 4.6: The **DispatchingService**. It's the client API. To the old architecture, the SSLDispatchingService component is added (in pink), which allows the client to open and communicate over a secure link.

a certificate presented by a neighbor can be considered trusted or not. In our specific distribution of SEC-REDS, two strategies exist:

- the *SubjectBasedTrust*, according to which the broker stores the list of characteristics that the certificate must have. In the specific case, the only characteristic is the subject, and
- the *CertificateBasedTrust*, in which the broker stores the list of valid certificates.

Figure 4.8 shows it.

If the certificate presented by the neighbor is not valid, the local *connection policy* is applied. Our specific implementation provides two types of connection policy:

- *BrutalCloseConnectionPolicy*, in which the connection is closed, without sending a message to the rejected client, and
- *UntrustedDegradeConnectionPolicy*, which degrades the neighbor to untrusted.

Figure 4.9 explains it.

To manage the *access control*, a new component is developed, the *AccessControlStrategy*, which directly interacts with the *SecurityManager*. As example, a generic *AccessControlStrategy* is realized, that, simply, applies the policy to messages/filters. The policy is stored into a configuration file. Figure 4.10 shows this component.

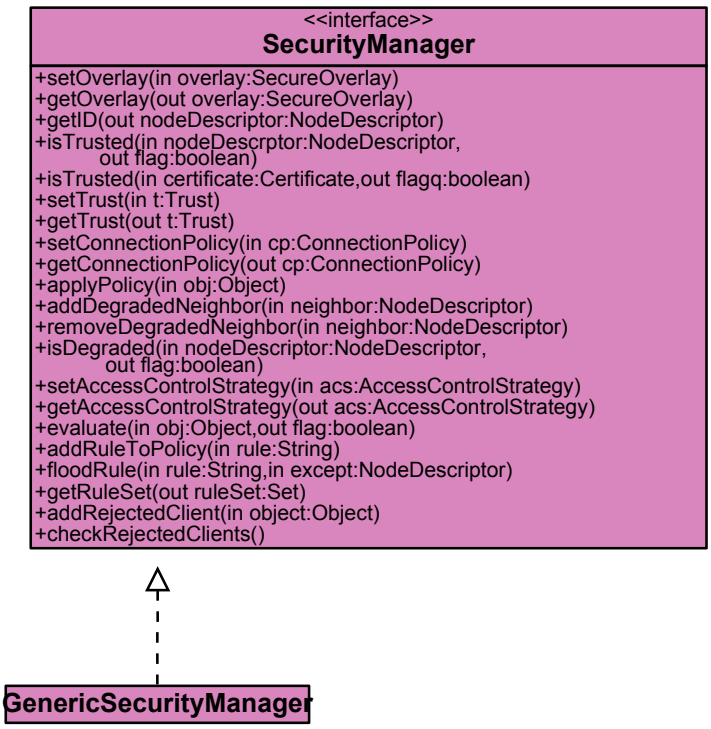

Figure 4.7: The **SecurityManager**. The SecurityManager is the central component of the new architecture. It interacts with the Trust, ConnectionPolicy and AccessControlStrategy objects, so it manages all the security aspects. The new architecture without the SecurityManager is like a sea without water.

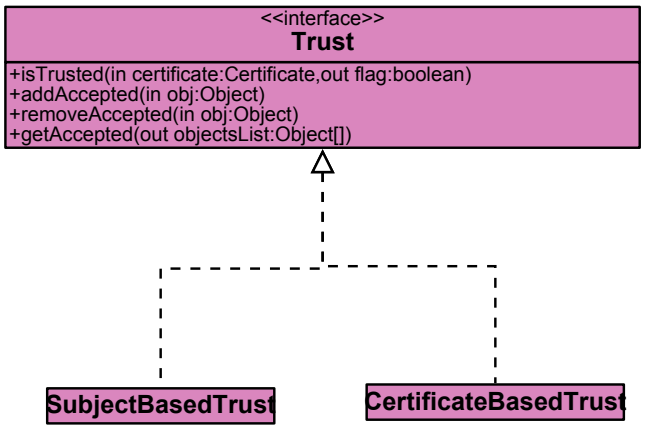

Figure 4.8: The **Trust**. This component interacts with the SecurityManager and establishes if a neighbor can be considered trusted. In the actual version of our implementation two strategies of trustworthiness are provided. These are shown in the figure.

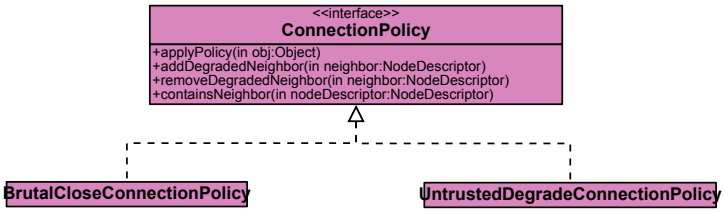

Figure 4.9: The **ConnectionPolicy**. This component interacts with the SecurityManager and establishes what strategy must be adopted if a neighbor can't be considered trusted. In the actual version of our implementation two strategies of trustworthiness are provided. These are shown in the figure.

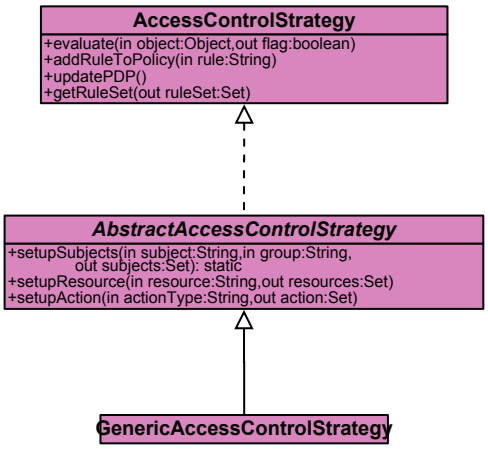

Figure 4.10: The **AccessControlStrategy**. This component interacts with the SecurityManager and manages all the access control. To the basic interface, an abstract class is added, the AbstractAccessControlStrategy, which defines basic operations, like building a subject, a resource, or an action. Then a GenericAccessControlStrategy is realized.

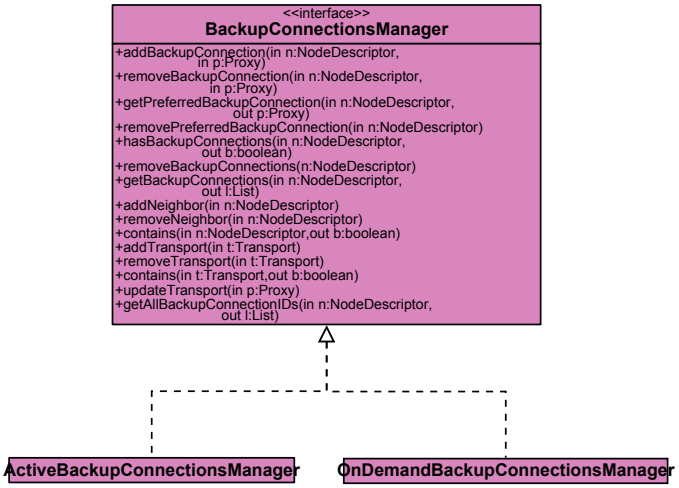

Figure 4.11: The **BackupConnectionsManager**.

4.3 The Backup Connections Manager

In order to support redundancy another component is added, that is the *BackupConnectionManager*, which works at the Transport layers and manages all the backup connections. It is shown in the Figure 4.11.

The actual version of SEC-REDS provides two implementations of the *BackupConnectionsManager*, the *ActiveBackupConnectionsManager* and the *OnDemandBackupConnectionsManager*. Those have been explained in the previous chapter.

Appendix A

A brief description of XACML

> Tutto ciò che può essere detto, può essere detto chiaramente.
>
> — Ludwig Wittgenstein (1889 - 1951)
>
> Austrian philosopher

A.1 The architecture

XACML [24] defines a general policy language used to protect resources as well as an access decision language.

Every enterprise has a need to secure resources accessed by employees, partners, and customers. For example, browser based access to portals which aggregate resources (web pages, applications, services, etc.) are typical in today's enterprises. Clients send requests to servers for resources, but before a server can return that resource it must determine if the requester is authorized to use the resource. This is where XACML fits in. XACML provides

a policy language which allows administrators to define the access control requirements for their application resources. The language and schema support include data types, functions, and combining logic which allow complex (or simple) rules to be defined. XACML also includes an access decision language used to represent the runtime request for a resource. When a policy is located which protects a resource, functions compare attributes in the request against attributes contained in the policy rules ultimately yielding a permit or deny decision.

Let consider the Figure A.1, that shows the *evaluation process* of a request sent by a client. The client sends a request (1) made of a tuple composed by a *subject*, a *resource* and an *action*. The subject identifies the originator of the request (e.g. John Smith), the resource represents the object that the subject wants to access (e.g. a file on a server), and the action specifies what the subject wants to do with that resource (e.g. download that file). The request is sent to the *Server Policy Enforcement Point* or simply *Policy Enforcement Point* (PEP, for short), which is the system entity that performs access control, by making decision requests and enforcing authorization decisions. The PEP sends the request to the *Policy Information Point* (PIP) (2), which is the system entity that takes from the received tuple the list of *attributes*. In fact, each member of the tuple (subject,resource,action) can have one or more *attributes*, characterizing that specific member. An attribute is made of *predicates*, that are statements about attributes whose truth can be evaluated. Then the PIP sends to the PEP the list of the obtained attributes. The PEP sends the request made of the tuple (subject,resource,action) and its attributes to the *Policy Decision Point* (PDP) (3), that is the system en-

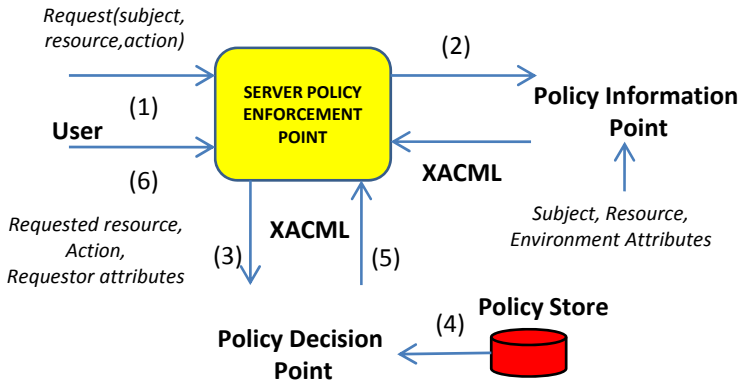

Figure A.1: XACML Architecture

tity that evaluates applicable policy and renders an *authorization decision*, which is the result of evaluating applicable policy. In particular, it consults the *Policy Store* (4), that maintains the access control policy and then it establishes whether the request can be accepted or not. The result of the decision can be *Permit* or *Deny*. Finally, it sends the result to the PEP (5), which communicates it to the user (6).

The XACML architecture is complete so an administrator can install it on its server. The only thing he must specify is the *access control policy*, which obviously may vary from a system to another.

An administrator creates policies in the XACML language. XACML defines three top-level policy elements: `<Rule>`, `<Policy>` and `<PolicySet>`. The `<Rule>` element contains a boolean expression that can be evaluated in isolation, but that is not intended to be accessed in isolation by a PDP. In particular, it is evaluated in a policy. A `<Rule>` is composed by three

fundamental elements, the <Subject>,the <Resource> and the <Action>. Through those elements, you can specify to which the rule must be applied. The rule has an effect, that is *Permit* or *Deny*.

The <Policy> element contains a set of <Rule> elements and a specified procedure for combining the results of their evaluation. In fact, more rules could result applicable to the same tuple (subject,resource,action). In this case, the procedure for combining the result specifies which rule must be applied. The procedure could be *first applicable* (the first rule found in the list is applied) or *permit overrides* (if a rule whose result is *Permit* is found, it is applied, independently from the other results). The <Policy> is the basic unit of policy used by the PDP, and so it is intended to form the basis of an authorization decision. A <Policy> can have also a <Target>, that specifies the set of requests, a policy is intended to evaluate. A <Target> is made of by three fundamental elements, the <Subject>, the <Resource> and the <Action>. Thus a policy using the target concept can establish a first level of visibility. In fact, among all the requests, only those matching with the target are further analyzed by that policy. Then, if among all the rules making the policy, at least one is found, a standard result (Permit or Deny) is provided, but if no rule is found, the result is *Not Applicable*. The <PolicySet> element contains a set of <Policy> or other <PolicySet> elements and a specified procedure for combining the results of their evaluation. It is the standard means for combining separate policies into a single combined policy.

If a client wants to build a request, he must create a request. This can be achieved using the XACML language. He creates a <Request> element, in which he can simply specify the tuple (subject, resource, action).

A.2 A simple example

Let consider the following example, in which a client tries to access to the server *Sample Server*. The action it wants to perform is of *login*. The XACML request is the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Request>
   <Resource>
      <Attribute
         AttributeId=
         "urn:oasis:names:tc:xacml:1.0:
         resource:resource-id"
         DataType=
         "http://www.w3.org/2001/XMLSchema#string">
         <AttributeValue>Sample Server</AttributeValue>
      </Attribute>
   </Resource>
   <Action>
      <Attribute
         AttributeId="Server Action"
         DataType=
         "http://www.w3.org/2001/XMLSchema#string">
         <AttributeValue>login</AttributeValue>
      </Attribute>
   </Action>
```

```
</Request>
```

The previous example specifies that a generic client wants to access the resource *Simple Server* to execute the action of login. In particular, both for the resource and the action he specifies *attributes*, characterized by an *attribute value* and an *attribute id*. Let suppose now that the server policy is the following:

```
<Policy PolicyId="SamplePolicy"
RuleCombiningAlgId=
"urn:oasis:names:tc:xacml:1.0:
rule-combining-algorithm:first-applicable">
<Target>
   <Subjects>
      <AnySubject/>
   </Subjects>
   <Resources>
      <Resource>
      <ResourceMatch
            MatchId=
            "urn:oasis:names:tc:xacml:1.0:
            function:string-equal">
            <AttributeValue
               DataType=
               "http://www.w3.org/2001/XMLSchema#string">
```

```
                SampleServer
            </AttributeValue>
            <ResourceAttributeDesignator
               DataType=
               "http://www.w3.org/2001/XMLSchema#string"
               AttributeId=
               "urn:oasis:names:tc:xacml:1.0:
               resource:resource-id"/>
         </ResourceMatch>
      </Resource>
   </Resources>
   <Actions>
      <AnyAction/>
   </Actions>
</Target>
<Rule
   RuleId="LoginRule" Effect="Permit">
   <Target>
      <Subjects>
         <AnySubject/>
      </Subjects>
      <Resources>
         <AnyResource/>
      </Resources>
```

```
      <Actions>
         <Action>
            <ActionMatch
               MatchId=
               "urn:oasis:names:tc:xacml:1.0:
               function:string-equal">
            <AttributeValue
               DataType=
                  "http://www.w3.org/2001/XMLSchema#string">
                     login
               </AttributeValue>
               <ActionAttributeDesignator
                  DataType=
                  "http://www.w3.org/2001/XMLSchema#string
                  AttributeId="Server Action"/>
            </ActionMatch>
         </Action>
      </Actions>
</Target>
</Rule>
<Rule RuleId="FinalRule" Effect="Deny" />
</Policy>
```

As we can see from the code, this policy applies only to clients that want to access to Sample Server. The first rule establishes that a client that wants to login to the server, can perform this action, while the second rule is applied to all the other clients, that want to perform other actions. In this case, they can't do them.

When the client sends the request to the server, this evaluates it and produces the result:

```
<Response>
   <Result>
      <Decision>Permit</Decision>
      <Status>
         <StatusCode Value=
            "urn:oasis:names:tc:xacml:
            1.0:status:ok" />
      </Status>
   </Result>
>/Response>
```

As expected, the server allows the client to login.

Appendix B

A simple case of study

> Example is the school of mankind, and they will learn at no other.
>
> —Edmund Burk (1729 - 1797)
>
> Irish philosopher

Let suppose that a patient wants to send his data to his physician and let suppose that he suffers of *angina pectoris*. Periodically, he sends his healthy conditions to his physician. But let suppose also that all the cardiologists wants to collect information about angina pectoris, without knowing the patient identity. An access control strategy can be used.

The patient, here named *John Patient*, can act as a *publisher* so he can define the following code:

```
KeyStoreData ksd = new KeyStoreData();
ksd.setKeystore("Patient","JKS","123456".toCharArray());
ksd.setRecord("patient","123456".toCharArray());
```

```
ksd.setSignature(null,"MD5withRSA");

XACMLMessage message = new XACMLMessage("John Patient",
null,"angina pectoris","publish",ksd);

String[] data = new String[3];
data[0] = "Name:John,Surname:Patient,born:4/2/1948,
country:Italy";
data[1] = "type of illness:steady angina";
data[2] = "number of pills in a day: if needed,
type of pill:Carvasin 10mg";
MessageContent mc = new MessageContent(data);
int[] cVis = {0};
mc.addMember("Andrew Smith", null);
mc.addMember("cardiologists",cVis);
message.setMessageContent(mc);

message.setRule("AllowAllCardiologists",
null,"angina pectoris" ,
"subscribe","cardiologists", "Permit");

DispatchingService es =
new SSLDispatchingService(dsAddr, dsPort);
es.open() throws Exception;
es.publish(message);
```

John Patient must present to the dispatching network an *Authorizations Certificate*. This certificate is recovered from his local *Key Store* (a Java Object in which local certificates are stored). In particular, he creates a `KeyStoreData` object, that contains all the data necessary to recover the Authorizations Certificate from the key store. the which in this case allows him to publish his publications. Then he creates a `XACMLMessage`, in which he specifies the asked operations (S,R,A) and the Authorizations Certificate (which will be recovered directly from the local key store). Then he create a `MessageContent` object, which contains the content of the message (in the example, he suffers of a type of angina, called *steady angina* and he must take a pill, named *Carvasin 10mg* only if needed) and the table of visibility. To add a member to that table, he uses the method `addMember(String member, int[] visibility)`, in which the vector *visibility* specifies the fields of the message content seen by that member. If it is null, all fields are available. Then he sets a rule for the message: he allows all the cardiologists to subscribe to his message. Finally he creates a `SSLDispatchingService` object and publishes the message.

Note that all cardiologists will receive the message, but with different visibilities. In fact, *Andrew Smith* will receive all the information, while a generic cardiologist only those about healthy conditions.

A generic cardiologists wanting to receive these informations, must present to the dispatching network a valid `AuthorizationsCertificate`, that classifies him as belonging to the cardiologists group. Then, he simply subscribes.

```
DispatchingService es =
```

```
new SSLDispatchingService(dsAddr, dsPort);
es.open();

KeyStoreData ksd = new KeyStoreData();
ksd.setKeystore("Cardiologist","JKS","123456".toCharArray());
ksd.setRecord("cardiologist","123456".toCharArray());
ksd.setSignature(null,"MD5withRSA");
es.subscribe(new XACMLFilter("Mark Cardiologist",
"cardiologists","angina pectoris","subscribe", ksd));

XACMLMessage m = (XACMLMessage)es.getNextMessage();
```

If the cardiologist is *Andrew Physician* he will receive the whole information.

Appendix C

Setting up a secure and robust broker

> If a man will begin with certainties, he shall end in doubt; but if he will be content to begin with doubts, he shall end in certainties.
>
> FRANCIS BACON (1561 - 1626)
> English philosopher

This appendix explains how a secure and robust broker could be built up. Note that in this context the term *object* and the term *component* have the same meaning. Firstly, the `Overlay` level is built up, setting up the `Transport`:

```
int TCPport = 8080;
int SSLport = 8081;
Transport normalTransport = new TCPTransport(TCPport);
```

```
SecureTransport secureTransprt = new SSLTransport(SSLport);
ComplexTransport transport =
new ComplexTransport(normalTransport,secureTransport);
```

In the example, two `Transport` are created, the `TCPTransport`, which listens at the port 8080, and the `SSLTransport`, which listens at the port 8081. Than the two `Transport` objects are joined using a `ComplexTransport` object, which manages them.

As next step, the `TopologyManager` component is created.

```
SecureTopologyManager topolMgr = new
SecureLSTreeTopologyManager(transport);
```

In the example, a `SecureLSTreeTopologyManager` is created, passing the `Transport` as parameter. This specific `TopologyManager` is an extension in terms of security of the `LSTreeTopologyManager` provided by the standard REDS architecture.

Now the overlay components are created, so the `Overlay` object can be created.

```
SecureOverlay overlay =
new SecureOverlay(topolMgr,transport);
```

The example shows the creation of a `SecureOverlay` object, that is an extension in terms of security of the `Overlay` provided by the standard REDS architecture.

After having created the Overlay level, the `SecurityManager` and the `BackupConnectionsManager` can be created.

```
SecurityManager secMgr =
new GenericSecurityManager(overlay);

BackupConnectionsManager bcm = new
OnDemandBackupConnectionsManager(secMgr);

transport.setBackupConnectionsManager(bcm);
```

The example employs a `OnDemandBackupConnectionsManager`. Then the Routing layer can be built up.

```
SecureRoutingStrategy routingStrategy = new
SecureSubscriptionForwardingRoutingStrategy();

Reconfigurator reconf =
new DeferredUnsubscriptionReconfigurator();

SubscriptionTable subscriptionTable =
new GenericTable();

SecureReplyManager replyMgr = new
SecureImmediateForwardReplyManager();

ReplyTable replyTbl = new HashReplyTable();

SecureRouter router =
```

```
new SecureRouter(secMgr,overlay);
```

The `RoutingStrategy` is created. The example shows the creation of a `SecureSubscriptionForwardingRoutingStrategy`, that is an extension in terms of security of the `SubscriptionForwardingRoutingStrategy` provided by the standard REDS architecture. Then the `Reconfigurator` object is built up, using the `DeferredUnsubscriptionReconfigurator`, implementation. Then the `SubscriptionTable` is created, using the `GenericTable` implementation. Also the `ReplyManager` and the `ReplyTable` are created. Note that the `SecureReplyManager` is an extension in terms of security of the `ReplyManager` provided by the standard REDS architecture. Finally the `Router` is created. The example shows the creation of a `SecureRouter` object, that is an extension in terms of security of the `Router` provided by the standard REDS architecture.

All those components must be joined, with the following strategy:

```
secureTransport.setSecurityManager(secMgr);

routingStrategy.setSecurityManager(secMgr);
routingStrategy.setOverlay(overlay);

reconf.setOverlay(overlay);
reconf.setRouter(router);

replyMgr.setReplyTable(replyTbl);
replyMgr.setSecurityManager(secMgr);
```

```
router.setSubscriptionTable(subscriptionTable);
router.setRoutingStrategy(routingStrategy);
router.setReplyManager(replyMgr);
router.setReplyTable(replyTbl);
```

As next step, all the security components (excluded the `SecurityManager` already defined) must be created.

```
Trust trust = new SubjectBasedTrust();
trust.addAccepted("John Patient");
```

The `Trust` object is created, using the `SubjectBasedTrust` implementation, in which you can specify the list of accepted subjects. In the example all the certificates in which *John Patient* is the subject, are considered as trusted. Then the `ConnectionPolicy` object is created.

```
ConnectionPolicy cp =
new UntrustedDegradeConnectionPolicy();
```

The example shows the use of the `UntrustedDegradeConnectionPolicy`, in which a neighbor is degraded to untrusted, if its certificate is not considered valid by the `Trust` object.

Eventually the `ConnectionPolicy` and the `Trust` components are joined to the `SecurityManager`:

```
secMgr.setTrust(trust);
secMgr.setConnectionPolicy(cp);
```

If you want to link a broker towards another broker, you simply invoke the following method:

```
String url = "reds-ssl:127.0.0.1:8081";
overlay.addNeighbor(url);
```

In this case, the broker tries to open a ssl connection towards the neighbor whose url is specified by the string url.

If you want to add a backup connection, you can invoke the following methods:

```
String defaultURL = "reds-ssl:155.34.10.3:8081";
BackupStrategy bs =
new BackupStrategy(defaultURL,neighbor);
bs.setTimeout(10000);
bs.setTentatives(2);
String backupURL = "reds-tcp:155.34.10.3:8080";
bs.addBackupConnection(backupURL);

bcm.addMasterBackupConnection(bs);
```

Firstly, the `BackupStrategy` is created, in which the default url and the neighbor are specified. The example shows that the timeout is 10 sec. and the number of tentatives 2. A backup connection is added, that is specified by the string backup url. Finally, the `BackupStrategy` is added to the `BackupConnectionsManager`.

Bibliography

[1] G. Muhl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. 2006.

[2] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. In *Computing Surveys*, volume 35. ACM, june 2003.

[3] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[4] Luca Rossellini. Progettazione e realizzazione di un sistema publish-subscribe con architettura cross-layer per reti ad-hoc. Master's thesis, University of Pisa, march 2003.

[5] *Computer and Information Sciences*, volume 2869/2003. Springer Berlin / Heidelberg, 2003.

[6] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symp. on Principles of Distributed Computing*, pages 53–61. ACM, 1999.

[7] G. Picco, G. Cugola, and A. Murphy. Efficient content-based event dispatching in presence of topological reconguration. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems*, pages 234–243. ACM, May 2003.

[8] E. Yoneki and J. Bacon. An adaptive approach to content-based subscription in mobile ad hoc net-works. MP2P-colocated with IEEE PERCOM, 2004.

[9] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notication service. In *ACM Trans. on Computer Systems*, volume 19, pages 332–383, August 2001.

[10] L. Mottola, G. Cugola, and G. Picco. A self-repairing tree overlay enabling content-based routing in mobile ad hoc networks. In *Technical report, Politecnico di Milano*, 2006.

[11] E. Yoneki and J. Bacon. Content-based routing with on-demand multicast. In *Proc. of the 3rd Int. Workshop on Wireless Ad Hoc Networking*. WWAN-colocated with IEEE ICDCS, 2004.

[12] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, aug 2001.

[13] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Sympo-*

sium on Principles of Distributed Computing, pages 219–227, Portland, Oregon, jul 2000.

[14] Antonio Carzaniga, David R. Rosenblum, and Alexander L. Wolf. Challenges for distributed event services:scalability vs. expressiveness. In *Engineering Distributed Objects '99*, Los Angeles, California, may 1999.

[15] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, dec 1998.

[16] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, Department of Computer Science, University of Colorado, aug 1998.

[17] Gianpaolo Cugola and Gian Pietro Picco. Reds: a reconfigurable dispatching system. In *SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware*, pages 9–16, New York, NY, USA, 2006. ACM Press.

[18] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 3940–3947, 2002.

[19] L. Fiege, A. Zeidler, A. Buchmann, R. Kilian-Kehr, and G. Muhl. Security aspects in publish/subscribe systems. *IEE Seminar Digests*, 2004(918):44–49, 2004.

[20] Lukasz Opyrchal and Atul Prakash. Secure distribution of events in content-based publish subscribe systems. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[21] D. Frey and A. Murphy. Maintaining publish-subscribe overlay tree in large scale dynamic networks. Technical report, Politecnico di Milano, www.elet.polimi.it/upload/frey, 2005. Submitted for publication.

[22] G. Cugola, G. Picco, and A. Murphy. Towards dynamic reconguration of distributed publish-subscribe systems. In *SEM02: Proc. of the 3rd Int. Workshop on Software Engineering and Middleware*, volume LNCS 2596, pages 187–202. Springer, May 2002.

[23] G. Cugola, D. Frey, A. Murphy, and G. Picco. Minimizing the reconguration overhead in content-based publish-subscribe. In *Proc. of the ACM Symp. on Applied Computing (SAC)*, pages 1134 –1140. ACM, 2004.

[24] Entrust Inc. Tim Moses, editor. *eXtensible Access Control Markup Language (XACML) Version 2.0*, 2005.

Index