Technical Report

UCAM-CL-TR-720
ISSN 1476-2986

Number 720

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A capability-based access control architecture for multi-domain publish/subscribe systems

Lauri I.W. Pesonen

June 2008

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Publish/subscribe is emerging as the favoured communication paradigm for large-scale, wide-area distributed systems. The publish/subscribe many-to-many interaction model together with asynchronous messaging provides an efficient transport for highly distributed systems in high latency environments with direct peer-to-peer interactions amongst the participants.

Decentralised publish/subscribe systems implement the event service as a network of event brokers. The broker network makes the system more resilient to failures and allows it to scale up efficiently as the number of event clients increases. In many cases such distributed systems will only be feasible when implemented over the Internet as a joint effort spanning multiple administrative domains. The participating members will benefit from the federated event broker networks both with respect to the size of the system as well as its fault-tolerance.

Large-scale, multi-domain environments require access control; users will have different privileges for sending and receiving instances of different event types. Therefore, we argue that access control is vital for decentralised publish/subscribe systems, consisting of multiple independent administrative domains, to ever be deployable in large scale.

This dissertation presents MAIA, an access control mechanism for decentralised, type-based publish/subscribe systems. While the work concentrates on type-based publish/subscribe the contributions are equally applicable to both topic and content-based publish/subscribe systems.

Access control in distributed publish/subscribe requires secure, distributed naming, and mechanisms for enforcing access control policies. The first contribution of this thesis is a mechanism for names to be referenced unambiguously from policy without risk of forgeries. The second contribution is a model describing how signed capabilities can be used to grant domains and their members' access rights to event types in a scalable and expressive manner. The third contribution is a model for enforcing access control in the decentralised event service by encrypting event content.

We illustrate the design and implementation of MAIA with a running example of the UK Police Information Technology Organisation and the UK police forces.

Marjalle

Acknowledgments

I would like to thank both my supervisor prof. Jean Bacon for her guidance and support both while I was considering to apply to Cambridge as well as during my time here as a PhD student. Jean also helped me find funding that allowed me to finish my studies, for which I am grateful to her. I am also indebted to Dr. Ken Moody for his advice and insights.

I should also express my thanks to all the Opera group members that I have had the pleasure of working with during the past four years. David Eyers especially has helped me tremendously by collaborating with me and proof-reading countless drafts of research papers and this dissertation. I am very grateful to David for all his help without which this dissertation might never have been finished. Eiko Yoneki has also provided invaluable advice concerning publish/subscribe systems. She is like a walking publish/subscribe research library. To András, Andy, Brian, Dan, David, Luis, Nathan, Peter, Salman, and Samuel: thanks for all the tea and biscuits.

A special thanks to Sriram who helped me discover my interest for programming languages that I did not even know I had. I expect this to have a lasting effect on my career.

I would like to thank my parents for supporting my academic endeavours. I know my moving abroad for the foreseeable future was not easy, but I have never received anything but support from them.

Finally, I want to try to express my gratitude to Sarah. She has tirelessly supported me through the final year of my PhD while I have been writing up my dissertation. Long nights and busy weekends have been the norm and the work has seemed never ending, yet she has always been there for me.

My work at Cambridge has been funded by the Engineering and Physical Sciences Research Council (EPSRC), Nokia Foundation, Jenny and Antti Wihuri Foundation, Tekniikan edistämissäätiö, and Helsingin Sanomain 100-vuotissäätiö.

Publications

- Lauri I. W. Pesonen and Jean Bacon. Secure event types in content-based, multi-domain publish/subscribe systems. In *SEM'05: Proceedings of the 5th international workshop on Software Engineering and Middleware*, pages 98–105. ACM Press, September 2005.

- Jean Bacon, David M. Eyers, Ken Moody, and Lauri I. W. Pesonen. Securing publish/subscribe for multi-domain systems. In Gustavo Alonso, editor, *Middleware'05: Proceedings of the 6th International Conference on Middleware*, volume 3790 of *LNCS*, pages 1–20. Springer-Verlag, November 2005.

- Lauri I. W. Pesonen, David M. Eyers, and Jean Bacon. A capabilities-based access control architecture for multi-domain publish/subscribe systems. In *SAINT 2006: Proceedings of the Symposium on Applications and the Internet*, pages 222–228, Washington, DC, USA, January 2006. IEEE Computer Society.

- Lauri I.W. Pesonen, David M. Eyers, and Jean Bacon. Access control in decentralised publish/subscribe systems. *Journal of Networks*, 2(2):57–67, April 2007.

- Lauri I. W. Pesonen, David M. Eyers, and Jean Bacon. Encryption-enforced access control in dynamic multi-domain publish/subscribe networks. In *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS'07)*, pages 104–115. ACM Press, June 2007.

- Luis Vargas, Lauri I. W. Pesonen, Ehud Gudes, and Jean Bacon. Transactions in content-based publish/subscribe middleware. In *DEPSA'07:Proceedings of the International Workshop on Distributed Event Processing, Systems and Applications*, page 68, Toronto, Canada, June 2007. IEEE Computer Society.

Contents

1 Introduction **19**
- 1.1 Multi-Domain Publish/Subscribe Systems . . . 21
- 1.2 Application Scenarios . . . 22
  - 1.2.1 Stock Ticker . . . 22
  - 1.2.2 Numberplate Monitoring . . . 24
- 1.3 Decentralised Access Control . . . 26
- 1.4 Research Statement . . . 26
- 1.5 Dissertation Outline . . . 27

2 Background **29**
- 2.1 Distributed Communication . . . 29
  - 2.1.1 Publish/Subscribe . . . 30
  - 2.1.2 Synchronous Request/Response . . . 33
  - 2.1.3 Asynchronous Messaging . . . 36
  - 2.1.4 Tuple Spaces . . . 38
- 2.2 Publish/Subscribe Subscription Models . . . 39
  - 2.2.1 Topic-Based Publish/Subscribe . . . 40
  - 2.2.2 Content-Based Publish/Subscribe . . . 41
  - 2.2.3 Type-Based Publish/Subscribe . . . 43
- 2.3 Decentralised Publish/Subscribe . . . 44
  - 2.3.1 Hermes . . . 45
- 2.4 Access Control . . . 47
  - 2.4.1 Mandatory Access Control . . . 48
  - 2.4.2 Discretionary Access Control . . . 49
  - 2.4.3 Role-Based Access Control . . . 52
- 2.5 Decentralised Trust Management . . . 53
  - 2.5.1 PolicyMaker . . . 54
- 2.6 Simple Public Key Infrastructure . . . 56
  - 2.6.1 Authorisation Certificates . . . 56
  - 2.6.2 Name Certificates . . . 58
  - 2.6.3 Group Subjects . . . 59

2.6.4 Threshold Subjects . . . 59
2.7 Summary . . . 60

3 Multi-Domain Publish/Subscribe Systems 61
3.1 A Multi-Domain Publish/Subscribe System . . . 61
3.2 Domains . . . 62
3.2.1 Sub-Domains . . . 63
3.2.2 Event Brokers . . . 63
3.2.3 Event Clients . . . 64
3.2.4 Access Control Service . . . 64
3.3 Principals . . . 65
3.4 The Coordinating Principal . . . 65
3.5 Transport Layer Security . . . 66
3.6 Threat Model . . . 68
3.7 Example Application . . . 69
3.8 Summary . . . 72

4 Secure Event Types 74
4.1 Event Type Definitions . . . 75
4.2 Secure Event Types . . . 77
4.2.1 Name Tuple . . . 78
4.2.2 Digital Signature . . . 80
4.3 Type Management . . . 80
4.3.1 Version Number . . . 82
4.3.2 Type Version Translation . . . 84
4.3.3 Authorisation Certificates . . . 87
4.4 Modifications Made to Hermes . . . 89
4.4.1 Type Storage . . . 90
4.4.2 API Changes . . . 91
4.4.3 Message Routing . . . 92
4.5 Performance . . . 92
4.6 Secure Names in Topic-Based Publish/Subscribe . . . 95
4.7 Related Work . . . 95
4.8 Summary . . . 96

5 Access Control 98
5.1 Access Control Model . . . 100
5.1.1 Authorising Domains . . . 100
5.1.2 Authorising Clients . . . 101
5.1.3 Authorising Event Brokers . . . 102
5.1.4 Authorising Sub-Domains . . . 105
5.2 Resources and Access Rights . . . 107
5.2.1 Event Service Access Rights . . . 107
5.2.2 Event Type Access Rights . . . 110
5.3 Verifying Authority . . . 115
5.3.1 Authentication . . . 116
5.3.2 Authorisation . . . 117

5.3.3 Verification in MAIA . . . 118
5.4 Delegating Root Authority . . . 119
5.5 Access Control in Topic-Based Publish/Subscribe . . . 120
5.6 Related Work . . . 121
5.7 Summary . . . 122

6 Policy Management 124
6.1 OASIS . . . 124
6.1.1 OASIS Policy in Our Example Scenario . . . 126
6.2 Access Rights Revocation . . . 127
6.2.1 Validity Period . . . 128
6.2.2 Certificate Revocation Lists . . . 129
6.2.3 SPKI On-Line Tests . . . 130
6.2.4 Active Revocation . . . 131
6.3 Distributing Validity Statements over Publish/Subscribe . . . 132
6.3.1 Request-Response over Publish/Subscribe . . . 133
6.3.2 State Caching . . . 134
6.3.3 Publishing Validity Statements . . . 134
6.4 Policy Evaluation at the Local Broker . . . 135
6.5 Distributing Capabilities . . . 136
6.5.1 Gathering Evidence . . . 136
6.5.2 Distribution Methods . . . 137
6.6 Related Work . . . 137
6.7 Summary . . . 138

7 Event Content Encryption 139
7.1 Event Level Encryption . . . 141
7.2 Attribute Level Encryption . . . 142
7.2.1 Emulating Attribute Level Access Control . . . 143
7.2.2 Restricted Attribute Values . . . 143
7.3 Encrypting Subscription Filters . . . 144
7.3.1 Coverage Relations with Encrypted Filters . . . 145
7.4 Avoiding Unnecessary Encryptions and Decryptions . . . 145
7.5 Implementation . . . 147
7.6 Key Management . . . 149
7.6.1 Secure Group Communication . . . 150
7.6.2 Key Refreshing . . . 151
7.7 Evaluation . . . 152
7.7.1 End-to-End Overhead . . . 152
7.7.2 Domain Internal Events . . . 154
7.7.3 Communication Overhead . . . 155
7.8 Related Work . . . 156
7.9 Summary . . . 159

8 Conclusions **160**
8.1 Contributions 160
8.2 Future Work 162
8.3 Summary 164

Bibliography **165**

List of Figures

1.1 A publish/subscribe systems consists of a number of publishers and subscribers and an event service decoupling the two from each other. 20
1.2 In decentralised publish/subscribe systems the event service is implemented as a network of event brokers. 20
1.3 A multi-domain publish/subscribe system consisting of three brokerage firms and one stock exchange. 23
1.4 A multi-domain publish/subscribe system consisting of the Metropolitan Police and the Congestion Control Service. 25

2.1 The event service and the use of asynchronous messaging decouples the publisher from the subscribers in time, space, and synchronisation. 32
2.2 The use of synchronous messaging and the lack of an intermediary couples the client tightly to the server. 34
2.3 In traditional message passing the use of asynchronous messaging achieves synchronisation decoupling between the message producer and the message consumer. 36
2.4 In a message queueing system the message broker decouples the producer from the consumers in time and space, but the consumers need to pull messages from the broker synchronously. 37
2.5 The *in* operation supports a many-to-one interaction model. 38
2.6 The *rd* operation in a Tuple Space allows many-to-many interaction between producers and consumers. 39
2.7 Subscribing to a topic in a topic hierarchy implies subscriptions to all sub-topics. 41
2.8 An SPKI authorisation certificate loop with three principals and two levels of delegation. 58

3.1 An overall view of our multi-domain publish/subscribe deployment 71

4.1 Detective Smith retrieves the Numberplate event type definition from a type registry and verifies its authenticity and integrity. 77
4.2 Translation to and from transit time events. 85
4.3 Translation to and from transit time events with attribute UIDs. 87

4.4 Verifying the name-signature link with and without a capabilities chain. 89
4.5 Subscription performance with and without certificate caching. 94

5.1 Capability 1 authorises the Met domain to subscribe to all attributes of the *Numberplate* event. Capabilities 2 and 3 delegate a subset of this capability to both the Met Broker and Detective Smith. 100
5.2 The blanket capability together with the capability issued to the Met domain authorises the broker to access type T_1. 103
5.3 The blanket capability together with the new capability issued to the Met domain authorises the broker to access type T_2. 104
5.4 An enclosing domain can group more privileged brokers and event clients into their own privileged sub-domains. 106
5.5 The principals and the capabilities form a tree where the principals are nodes and the capabilities are vertexes. 109

7.1 In order to emulate attribute level encryption with event level encryption the publisher must publish independent events for all subscriber groups. 143
7.2 Caching decrypted data can increase efficiency when delivering an event to a peer with similar privileges. 146
7.3 Node addressing is effectively random, therefore the rendezvous node for a domain internal type can be outside of the domain that owns an event type. . . . 147
7.4 The EAX mode of operation. 148
7.5 The steps involved for a broker to be successful in joining a key group. 150
7.6 Key refreshes can be delayed based on the validity times of the broker's authority. 152
7.7 The end-to-end test setup. 153
7.8 The end-to-end throughput of events with plaintext events, event level encryption, and attribute level encryption. 154
7.9 The end-to-end throughput of events with plaintext events, event level encryption, and attribute level encryption when plaintext caching is enabled. 155
7.10 The average number of hop counts when emulating attribute level encryption with event level encryption and multiple sub-types ($\log$ scale). 156

List of Tables

1.1 The *StockTicker* event. 22
1.2 The *Numberplate* event. 24

2.1 The basic publish/subscribe API is very simple, consisting of only five operations. 31
2.2 The tuple space API of three operations used to write, read, and consume tuples. 38
2.3 Common classification labels in decreasing order of access. 48
2.4 An access control matrix representing files in a Unix system. 50
2.5 An access control list represents one column of the access control matrix. . . . 51
2.6 A capability represents a group of cells on one row of the access control matrix. 52

4.1 A Hermes-style event type definition. 75
4.2 A secure event type definition. 78
4.3 The *Numberplate* event type definition with a version number and attributes with unique identifiers. 88
4.4 A secure event type definition with a *credentials* field. 88
4.5 The Hermes event client API. 91
4.6 The MAIA event client API. 92
4.7 The time in microseconds spent on 5-tuple reductions on RSA signature verifications. 93
4.8 The time in microseconds spent on processing a subscription request at the local broker for plain types and signed types when the type cache is enabled. 93

CHAPTER 1

Introduction

Large-scale, multi-domain publish/subscribe systems require an access control mechanism in order to be deployable. This dissertation proposes a discretionary access control architecture for decentralised publish/subscribe systems spanning multiple independent administrative domains.

Very large-scale distributed systems, that cover large geographic areas and consist of a large number of nodes, are commonplace in today's networked world. The pervasiveness of the Internet and ever more affordable networking equipment facilitate the building of increasingly large systems with relatively low costs. Large-scale distributed systems are being built between (i) organisations (e.g. supply-chain management, work-flow management), (ii) individuals (e.g. instant messaging, IP telephony, and especially peer-to-peer applications), and finally between (iii) organisations and individuals (e.g. RSS feeds, content delivery, and AJAX-based web applications). Publish/subscribe has emerged as a scalable communication paradigm for large-scale distributed systems where traditional paradigms, e.g. request-response and simple asynchronous message passing, have struggled. Publish/subscribe is not a silver bullet to be used in all distributed systems. For example, the publish/subscribe interaction model, where publishers push data to the subscribers, is not suitable for all applications, and the lack of a reply channel makes the implementation of some distributed applications cumbersome.

A publish/subscribe system decouples event producers, i.e. *publishers*, from event consumers, i.e. *subscribers*, by introducing an abstract *event service* between the communicating parties (See Figure 1.1). The event service is responsible for delivering published events from publishers to all subscribers who have registered their interest in the given event. The decoupling of publishers from subscribers combined with asynchronous messaging allows publish/subscribe systems to scale in size both with respect to the number of nodes as well as the geographic distances between nodes (i.e. increasing network latency).

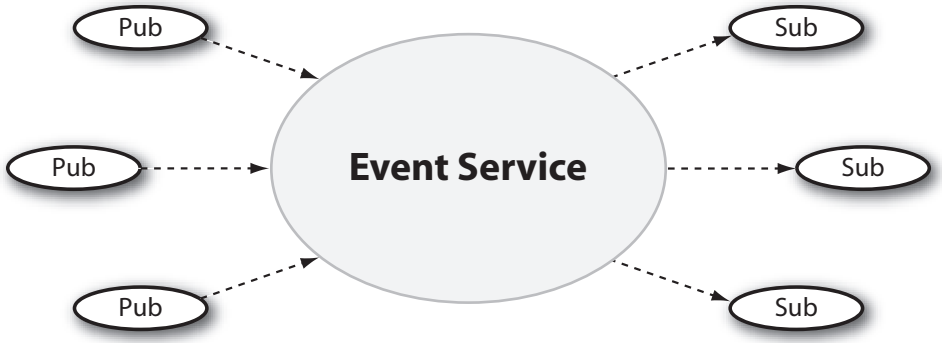

Figure 1.1: A publish/subscribe systems consists of a number of publishers and subscribers and an event service decoupling the two from each other.

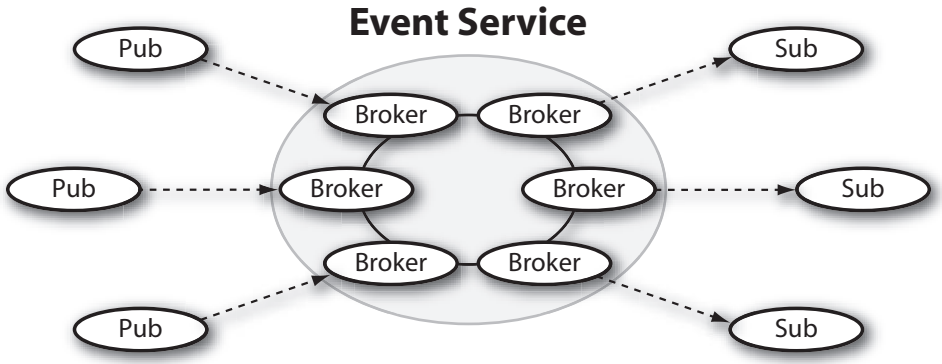

Figure 1.2: In decentralised publish/subscribe systems the event service is implemented as a network of event brokers.

Modern, highly scalable publish/subscribe systems implement the event service as a decentralised network of interconnected *event brokers* (See Figure 1.2). *Event clients*, i.e. publishers and/or subscribers, connect to a *local broker* in order to access the decentralised event service. The broker network then routes publications from publishers to subscribers. A decentralised event service enhances system scalability, fault-tolerance, and load balancing in a large-scale setting by distributing system load among all the participating brokers and by providing redundant routes between publishers and subscribers. The main motivation for deploying a decentralised event service is to be able to service a large number of event clients. Therefore, when the number of event clients in a publish/subscribe system grows past a certain point the event service must be decentralised across multiple event brokers.

In the past most publish/subscribe oriented research has concentrated on efficient routing algorithms, content-based filtering, and composite event detection in a single-domain environment. Relatively little research has been done with respect to security in publish/subscribe systems, especially in a multi-domain setting, yet we believe scalable access control to be a prerequisite for large-scale publish/subscribe systems to be widely adopted and deployed.

1.1 Multi-Domain Publish/Subscribe Systems

Publish/subscribe systems have been advocated especially for large-scale systems where the event service covers a large geographic area, because the publish/subscribe communication paradigm performs extremely well under high latency conditions compared to other alternatives.

We expect that large-scale publish/subscribe systems will be, in most cases, formed by multiple cooperating domains, where the domains represent separate organisations, sub-domains of a single organisation, or a mix of the two. It is unlikely that a single organisation would deploy a large-scale publish/subscribe system with hundreds of brokers spanning a large geographic area as a single domain. Instead the publish/subscribe system would span multiple independent administrative domains (e.g. business units or divisions) in the organisation. Examples of such systems include commercial applications, e.g. in the banking world or logistics systems, large-scale public sector systems, e.g. in the health care and law enforcement sectors, and sensor-based systems, e.g. city-wide sensor networks.

The domains cooperate to form a publish/subscribe infrastructure that is shared among all the domains (cf. the Internet email infrastructure). The motivation for domains to share the infrastructure is three-fold: (i) the shared publish/subscribe system reaches a wider geographic area and more users; (ii) the shared publish/subscribe system is more tolerant of node and network link failures, because of redundant nodes and routes, both without additional infrastructure expenses; and (iii) the shared infrastructure allows domains to implement applications with each other.

Figure 1.3 shows a publish/subscribe system consisting of four independent administrative domains: three brokerage firms and a single stock exchange. The four domains cooperate together in order to share the infrastructure as well as applications running on that infrastructure. The stock exchange scenario will be described in more detail in §1.2.1.

The domains are expected to deploy both their public and private publish/subscribe applications on the shared publish/subscribe system. Again the motivation for deploying all applications on the same publish/subscribe system is based on geographic reach and fault tolerance, as above. The deployed applications can be freely accessible to all domains (i.e. *public*), access can be limited to one or more other domains (i.e. *shared*), or the applications can be domain-internal only (i.e. *private*). We will limit our discussion to private and shared applications that are more interesting with respect to access control.

In order to facilitate the deployment of shared and private applications the publish/subscribe system must provide an access control mechanism that can be used to prevent unauthorised parties from accessing protected applications. The application owner must be able to specify in an access control policy who is authorised to issue publications and subscriptions in the context of a given application.

Attribute Name	Description
`time`	Time of the sale
`stock`	Name of the stock
`shares`	Number of shares sold
`price`	Price per share
`seller`	Brokerage firm selling the shares
`buyer`	Brokerage firm buying the shares

Table 1.1: The *StockTicker* event.

1.2 Application Scenarios

In order to motivate the necessity of access control in a multi-domain environment we present two application scenarios. Both scenarios involve multiple domains and underline the need for an expressive access control system that is able to span domain boundaries. The second example application, *numberplate monitoring*, is used as a running example throughout the rest of the dissertation.

1.2.1 Stock Ticker

The stock ticker presents a good example of a publish/subscribe application with access control needs on multiple levels that can be deployed in a multi-domain environment. A stock ticker reports each stock trade that happens in a stock exchange. The latest trade of a share determines the current price for that share, which is then reported to investors.

In this scenario a stock exchange acts as the event publisher. There can be any number of stock exchanges as publishers in the system, but for this example we will assume only one stock exchange. The exchange publishes events for each stock trade that takes place in the exchange. As shown in Table 1.1, the event includes the time of the sale, the name of the stock, the number of shares sold, the price for one share in this transaction, and the names of the buying and selling brokerage firms.

Private investors typically use a brokerage firm for their stock trades, because it is very expensive to interface directly with the stock exchange and the number of trades conducted by a single individual do not justify the cost. In our scenario the brokerage firms act as service providers for the private investors: the brokerage firm provides the private investor access to the publish/subscribe system. In terms of event clients and brokers, the investors act as event clients that connect to the event brokers provided by the various brokerage firms, and the brokerage firms and the stock exchange form together the broker network that implements the event service, as shown in Figure 1.3. Lastly, the stock exchange implements an event publisher that connects to the event broker provided by the Stock Exchange and publishes trade events.

The exchange publishes a number of event flows covering the sales of different types of financial instruments, e.g. shares, commodities, and derivatives, that all produce their own flow

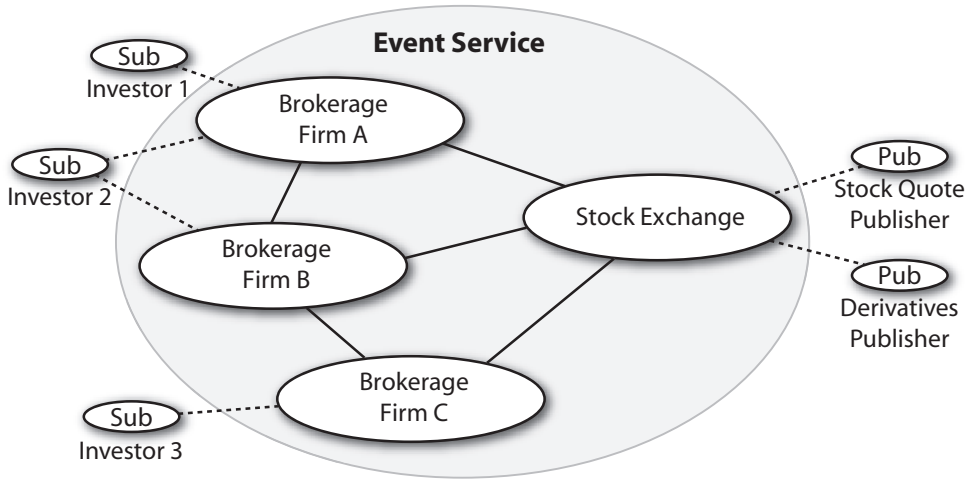

Figure 1.3: A multi-domain publish/subscribe system consisting of three brokerage firms and one stock exchange.

of trade events. The exchange charges the brokerage firms a fee for access to the different event flows. Each brokerage firm is free to choose which event flows it needs access to. For example, a brokerage firm might want to access only the shares event flow, but not the event flows for derivatives or commodities trades.

The brokerage firms charge their customers, the private investors, for access to the event streams available to the brokerage firms. A customer chooses her brokerage firm based on which event flows she wants to access and which event flows are available from each brokerage. E.g. if an investor wants access to derivatives events, she would pick a brokerage firm that can provide that access.

In this example each brokerage firm and stock exchange represent their own domains. The customers are event clients that access the publish/subscribe system via their brokerage firm. Effectively the customers are members of the domain of their brokerage firm. The stock exchange grants other domains access to events it publishes (inter-domain access control). The brokerage firms grant members of their own domains access to the published events (intra-domain access control). We can assume that the access control policy at the stock exchange remains relatively static, because brokerage firms rarely change the set of services they have subscribed to from the stock exchange and new brokerage firms enter the system very infrequently. The access control policies at the numerous brokerage firms, on the other hand, are in a constant state of churn, when brokerage accounts are opened and closed as clients join and leave the brokerage firm. The two layer access control approach accommodates this dichotomy of requirements very well.

Attribute Name	Description
`time`	Time of numberplate sighting
`numberplate`	Sighted numberplate
`location`	Location of sighting, i.e. the location of the camera

Table 1.2: The *Numberplate* event.

1.2.2 Numberplate Monitoring

The city of London in the UK introduced the London Congestion Charge on 17th February 2003 [Tra07]. In the scheme a vehicle must pay a fee for entering a congestion controlled area in central London. The fee must be paid before the vehicle actually enters the monitored area. Payment can be made on the web, over SMS, or at specific pay points. The charge is enforced by CCTV cameras that monitor vehicles going in and out of the congestion charge area. The cameras take pictures of numberplates, which are then run through numberplate recognition software. The resulting list of numberplates is compared to a list of numberplates that have paid the congestion control fee for that day. If a numberplate is not present in the database for that day, the owner of the vehicle is fined for not paying the fee on time.

We use the congestion charge scenario as an example application for motivating the need for access control in multi-domain publish/subscribe systems. In our example the monitoring and payment models differ from the system currently in use in London in two ways: (i) we assume that vehicle owners are sent a bill or their pay-as-you-go accounts are debited when their vehicle is seen inside the congestion controlled area, and (ii) that the CCTV cameras are able to perform numberplate recognition internally and act as publishers in a publish/subscribe system publishing *Numberplate* events for each recognised numberplate. We also assume that the Metropolitan police force is able to get access to the numberplate events based on a court order.

In our example a CCTV camera publishes a *Numberplate* event when it has recognised the numberplate of a vehicle entering the congestion controlled area. As shown in Table 1.2, the event contains the numberplate of the vehicle, the location of the CCTV camera, and a publication timestamp specifying the time when the vehicle was sighted.

A Congestion Control Service (CCS) billing office subscribes to *Numberplate* events. Each published *Numberplate* event will be delivered to the data centre, which processes the event by comparing the numberplate to a database of numberplates that have already been charged the congestion charge for that day. If the numberplate is not in the database, the vehicle owner is sent a bill or her pay-as-you-go account is debited.

The numberplate monitoring service has its uses outside of congestion control. The *Numberplate* events can be used to get notifications of sightings of a specific vehicle, which would be useful in criminal investigations where a specific numberplate is related to a case investigated

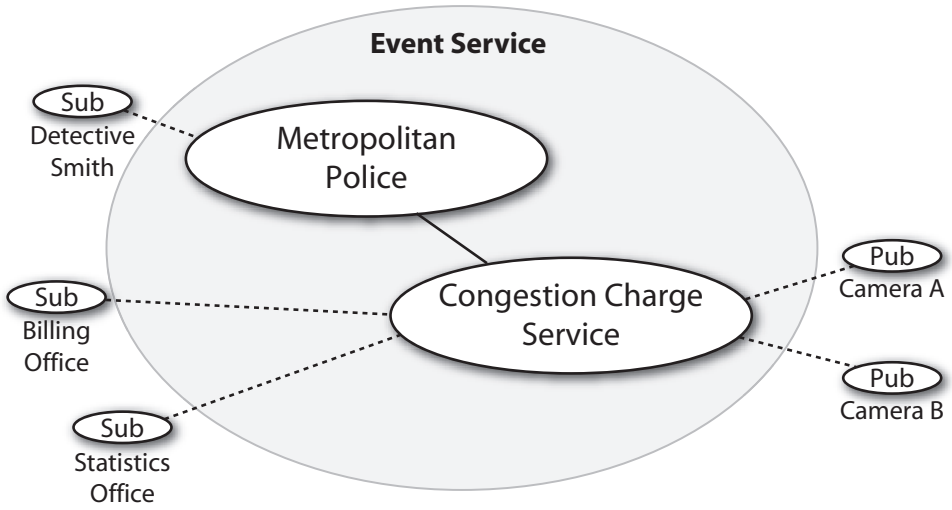

Figure 1.4: A multi-domain publish/subscribe system consisting of the Metropolitan Police and the Congestion Control Service.

by the police. Another example would be gathering traffic statistics from the congestion control cameras.

Figure 1.4 illustrates the publish/subscribe system consisting of the Metropolitan Police and the CCS domains. The cameras act as *Numberplate* event publishers, connected to the CCS domain. Detective Smith, the billing office and the statistics office all act as subscribers to the *Numberplate* events[1].

Free access to numberplate sightings presents a massive privacy concern. Therefore, in order to protect the privacy of vehicle owners and drivers, access to the events must be controlled according to an access control policy. The three different applications need access to different attributes of the numberplate events. The congestion control service needs to access the numberplate and timestamp fields in order to be able to send a bill to the owner of the vehicle for entering the congestion controlled area, but it does not need to know the specific location of the sighting. The police needs access to all the attributes of a numberplate event to be able to track down a given vehicle, but she does not need access to all numberplate events, instead her access can be limited to those events related to a specific numberplate. And finally statisticians need to know about locations and the time of each sighting, but they have no need for the numberplate information. An access control architecture for a multi-domain publish/subscribe system should be able to cater for each of these three scenarios.

We will use the numberplate monitoring application as a running example throughout this dissertation. See §3.7 for a more detailed description of the example environment and the application.

[1]While our application scenario is fictional, the anti-terror officers in the Metropolitan police in London did get access to real-time congestion charge data recently [New07]

1.3 Decentralised Access Control

A multi-domain publish/subscribe system is inherently decentralised. All domains are considered equal and there is no central party that can be trusted by all participants to control access to their applications. On the one hand the lack of a central authority enables all domains to implement and deploy their own applications on the shared infrastructure as equals. On the other hand it prevents implementing a relatively simple centralised access control architecture, because there is no trusted central authority to host it.

We propose a decentralised access control architecture where application owners are responsible for defining and managing application specific access control policies, and the access control decision making and enforcement is decentralised over all the domains in the system and their event brokers.

Application owners delegate access rights to domains. Each domain further delegates those rights to publishers and subscribers that are its members. The domains are able to implement domain-specific access control policies.

A decentralised model can also be more scalable: it avoids single points of failure in the verification infrastructure; it is more manageable in a multi-domain setting; and it will afford better verification performance by localising access control checks to event brokers. In some cases, though, the policy management requires that access rights must be immediately revocable, in which case decentralised access control models tend to degrade to making queries to a central revocation server for each access control decision.

1.4 Research Statement

In this dissertation we argue that large-scale publish/subscribe systems will most likely be formed by multiple independent administrative domains. The presence of more than one domain necessitates an access control mechanism that is able to enforce an access control policy and protect the confidentiality and integrity of event content in a shared infrastructure.

The main contribution of this work is the design and prototype implementation of a decentralised access control architecture for multi-domain publish/subscribe systems called MAIA. MAIA provides:

Unique Names MAIA creates unique type and topic names by prefixing the human-readable name with the name owner's identity as a cryptographic public key. The globally unique public key defines a globally unique namespace inside which the owner of the public key is free to define new names. Assuming that the name owner is able to avoid name collisions inside its own namespace and that the public key crypto system is not broken we can guarantee that names in MAIA are unique.

Type Authenticity and Integrity Type definitions are digitally signed by the type owner when they are deployed. The signature can be verified with the type owner's public key that is part of the event type name, i.e. a type definition is self-certifiable. By verifying the signature on the type definition, the verifier (i.e. event client or broker) can ascertain the integrity and authenticity of the type definition.

Network-Level Access Control MAIA enables a coordinating principal in the multi-domain publish/subscribe system to control who is able to join the publish/subscribe system on an infrastructure level. Only those domains that have been authorised by the coordinating principal (directly or indirectly by a delegate of the coordinating principal) are able to join and access the shared publish/subscribe system.

Application-Level Access Control MAIA enables type owners to control who is able to use (i.e. publish or subscribe to) a type. A domain must be authorised by the type owner (directly or indirectly by a delegate of the type owner) to use a given type. This work is equally applicable to topic based publish/subscribe systems.

Event Content Encryption MAIA enforces event-level access control in the event service by encrypting the content of published events. Encrypting the event content prevents an event broker that is routing the event from accessing the event content unless it has been granted access to the appropriate encryption keys.

1.5 Dissertation Outline

The remainder of this dissertation is organised as follows:

Chapter 2 provides an introduction to the background necessary to understand access control in large-scale publish/subscribe systems. The chapter first discusses publish/subscribe as a messaging paradigm, and more specifically decentralised publish/subscribe systems. The second part of the chapter discusses various access control models and then moves on to looking at decentralised trust management and specifically at the *simple public key infrastructure* (SPKI).

Chapter 3 describes what we understand domains to be and what a multi-domain environment is expected to look like. The first part of the chapter introduces the various components present in a domain, and what are the responsibilities of an access control service in a domain. The second part describes how a coordinating principal forms a multi-domain environment by inviting domains to join a shared infrastructure. We finish the chapter with a detailed description of the vehicle congestion control example application that will be used throughout this dissertation to motivate our work.

Chapter 4 provides a foundation for a decentralised access control system. The chapter presents a scheme for secure names, verifiable event type definitions, and a mechanism for

updating type definition in a live publish/subscribe system. Secure names and verifiable event type definitions enable us to reference event types securely from access control policy.

Chapter 5 presents our decentralised access control architecture for multi-domain publish/subscribe systems. The chapter begins with a description of capability-based access control and access right delegation. The second part of the chapter describes the access control mechanism for network-level and application-level access control.

Chapter 6 discusses access control policy management, various approaches to access right revocation, and how to deliver capabilities and credential validity statements in the publish/subscribe system.

Chapter 7 addresses the issue of enforcing access control policy inside the publish/subscribe event service. The architecture presented in Chapter 5 provides access control at the edge of the publish/subscribe system's event service, but it does not enforce access control while the event is being routed through the system possibly via untrusted intermediaries.

Chapter 8 summarises the work presented in this dissertation and outlines future work on decentralised publish/subscribe systems in the Opera group.

CHAPTER 2

Background

In this chapter we introduce the concepts and related work that the access control model presented in this dissertation builds on. The first part of the chapter concentrates on communication in distributed systems and the publish/subscribe interaction paradigm. We discuss the various distributed communication paradigms and compare them to the publish/subscribe paradigm in §2.1. In §2.2 we introduce the three most common subscription models used in publish/subscribe today. Finally, in §2.3 we discuss decentralised publish/subscribe systems, which are at the heart of multi-domain publish/subscribe systems and the basis of this dissertation.

The second part of this chapter introduces the reader to access control concepts that are relevant to this work. In §2.4 we introduce basic access control concepts and discuss the background of access control research. We move on to cover decentralised trust management systems in §2.5, and finish the chapter with an in depth view of the Simple Public Key Infrastructure, which we use as an underlying access control mechanism in MAIA, in §2.6.

2.1 Distributed Communication

A distributed system consists of a number of nodes that are connected to each other over a computer network, e.g. a *local area network* (LAN), a private *wide area network* (WAN), or the Internet. These nodes implement one or more distributed applications by running concurrently and communicating with each other over that network. Coulouris et al. define a distributed system as "one in which components located at networked computers communicate and coordinate their actions only by passing messages" [CDK01].

The communication between nodes can be implemented in a number of different ways. Various communication paradigms, that formalise the node to node communication in a distributed

system, have been proposed in the past. We will discuss *publish/subscribe*, *synchronous request/response*, *asynchronous message passing*, and *tuple spaces* in more detail in the following sections.

The various paradigms differ from each other in the type of abstraction provided for the programmer, in the type of abstraction used in the implementation, and in the level of coupling between communicating nodes. Notice that the communication abstraction offered to the programmer and the underlying implementation are orthogonal, e.g. the programming model can provide an asynchronous messaging API to the programmer while the underlying implementation is based on synchronous remote procedure calls and vice versa.

In this dissertation we concentrate on large-scale distributed systems. Therefore we are first and foremost interested in the scalability properties of the various communication paradigms. The next section will describe the publish/subscribe interaction paradigm in detail. The following sections will compare various alternative interaction paradigms to publish/subscribe and discuss the differences from a scalability point of view.

2.1.1 Publish/Subscribe

In the publish/subscribe interaction paradigm event producers, i.e. *publishers*, publish events, which are delivered to interested event consumers, i.e. *subscribers*. The subscribers declare their interests in the form of *subscriptions*: a subscriber will be notified of all events that match its subscription. We will discuss the various subscription models in more detail in §2.2.

Each publication is delivered to all subscribers with matching subscriptions. If there are no matching subscriptions, the publication is not delivered to any subscriber (We will discuss more expressive subscription models in §2.2). Multiple publishers are able to publish to the same subscriber group. Therefore, publish/subscribe is said to implement a many-to-many communication model.

A publish/subscribe system consists of *event clients*, that can be publishers, subscribers, or both, and an *event service*. The event clients connect to the event service in order to access the publish/subscribe system. The event service is responsible for delivering publications to subscribers by matching the publication to the active subscriptions.

The implementation of the event service depends on the size and type of the system being supported. Small systems run on a single server where all the event clients and the event service are running on a single node, possibly even as a single application. Medium sized systems run each event client and the event service on separate nodes. Finally, very large systems need to implement the event service as a decentralised service over a set of nodes. We will discuss decentralised publish/subscribe systems in §2.3.

The API exported by the event service is very simple (See Table 2.1). A subscriber calls the `subscribe` operation on the event service to create a new subscription. The `unsubscribe` operation allows the client to cancel an earlier subscription. The publishing API consists of one

Function	Description
advertise()	advertise an event type / topic
unadvertise()	undo a previous advertisement
publish()	publish an event of a type / topic
subscribe()	subscribe to events
unsubscribe()	cancel an existing subscription

Table 2.1: The basic publish/subscribe API is very simple, consisting of only five operations.

or two operations depending on the underlying publish/subscribe system. All publish/subscribe systems support a `publish` operation for publishing events to the event service. Some publish/subscribe systems also provide an `advertise` operation, which is used in decentralised systems to create routing state and in other systems to provide information to potential subscribers on the types of events that are being published in the system.

The event service isolates the event clients from each other, as shown in Figure 2.1, and stores and maintains the event routing information (i.e. the subscriptions and advertisements). The indirection provided by the event service allows the subscribers to be decoupled from the publishers in three dimensions [EFGK03]:

Synchronisation decoupling

Communication between two nodes in a distributed system can be either synchronous or asynchronous. With synchronous communication a client node makes a request to a server node that then replies to that request (See §2.1.2 for a more detailed discussion on *request/response*). Because the communication is synchronous, both nodes "meet in time", i.e. both parties are involved in handling the message at the same point in time. Typically the client has to wait for the response from the server before it can continue its execution. This means that the client thread has to block for the time it takes for the request to be delivered to the server, the server to handle the request, and the response to be delivered back to the client. The fact that the requester blocks for the duration of the synchronous interaction hinders the scalability of the distributed system. Both the number of interacting nodes as well as the link latencies between nodes degrade the performance of a synchronous system, because client nodes block for increasingly longer periods of time.

The publish/subscribe interaction paradigm avoids this problem by using an asynchronous method of communication. A publisher submits a publication to the event service. After that the event service is responsible for delivering the publication to the subscribers. The publisher does not have to wait for the publication to be delivered to the subscribers before it can continue its execution. Typically the underlying implementation allows the application to continue its execution immediately while the network communication is handled in the background. That is, the publishing application pushes the publication onto a *send queue*, which is then handled

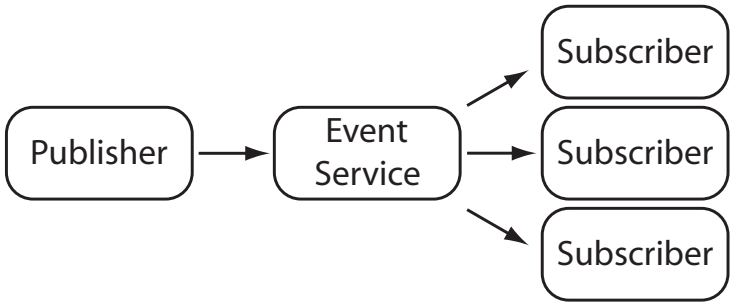

Figure 2.1: The event service and the use of asynchronous messaging decouples the publisher from the subscribers in time, space, and synchronisation.

in the background by another thread of execution. The same is true for the other messages in the publish/subscribe system, i.e. subscriptions and advertisements.

Spatial decoupling

In order for two nodes to be able to communicate with each other, messages from a sender must reach the recipient. If the sender is required to know the specific recipient (e.g. the recipient's name or IP address) in order to send a message, the two nodes are considered to be tightly coupled to each other in space. Space coupling affects the scalability of a distributed system by requiring nodes to have knowledge of all of their communication parties. This increases the memory consumption of applications and it is difficult to maintain that state in large-scale systems with high node churn rates, like the multi-domain systems we are envisioning.

We can alleviate this tight coupling to some extent by introducing a level of indirection between the nodes. For example, a *name service* that maps names to addresses or objects allows the recipient to change this mapping without it affecting the sender. Examples of such implementations include the *domain name service* (DNS) used in the Internet to map hostnames to IP addresses, and name services that are used in various request/reply style middleware products to map names or interfaces to services or implementations, e.g. object registries in RMI and CORBA (See §2.1.2).

In the publish/subscribe model the event service conveniently hides all the event clients from each other. Instead of publishing an event to specific subscribers, the publisher submits the publication to the event service and the event service delivers it to the subscribers. This means that only the event service needs to know of all the participating nodes. We will discuss in §2.3 how this state can be decentralised in large-scale publish/subscribe systems by implementing the event service as a network of *event brokers*.

Temporal decoupling

Two nodes are said to be coupled in time if both nodes need to be active at the same time in order to be able to communicate with each other. An active node in this case means a node that is connected to the distributed system and executing the distributed application. In a typical client/server model both the client and the server must be running at the same time in order to be able to communicate with each other. If the client were to make a request to a server that was not connected to the network, or was not executing the server application, the request would fail.

In message based systems a message broker can be used to store messages while the recipient is inactive or disconnected from the system. The message broker stores the message while the recipient is not available thereby hiding this fact from the sender.

Decoupling the nodes from each other in time allows the nodes to connect and disconnect from the distributed system without affecting its overall functionality. In a large-scale distributed system it is expected that nodes join and leave the system frequently either of their own volition or because they have been started or they have crashed. The fact that the churn does not affect the other nodes in the system allows the system to scale better.

In a publish/subscribe system the decoupling provided by the event service allows publishers and subscribers to join the system without there having to be a counterpart for them. For example, a publisher can join the publish/subscribe system and publish events when there are no subscribers with matching subscriptions. Similarly a subscriber can subscribe to events when there are no publishers in the system.

A publish/subscribe system can also provide disconnected operation where the event service caches events for a client that have left the system temporarily and replays those events back to the client when it rejoins (See [SAS01, PCM03, FGKZ03, CMPC04, MUHW04]).

2.1.2 Synchronous Request/Response

One of the earliest proposed paradigms for communicating between nodes in a distributed system was the *remote procedure call* (RPC). An early form of RPC was proposed by James White in an *Internet Engineering Task Force* (IETF) *Request for Comments* (RFC) titled *A High-Level Framework for Network-Based Resource Sharing* [Whi76]. White's goal was to standardise a communication protocol and an early form of middleware that would allow developers to reuse the communication layer when implementing distributed applications like the *file transfer protocol* (FTP) [PR85] for IP networks. Birrell and Nelson published their seminal paper on implementing an RPC mechanism eight years later [BN84], which provided the cornerstone for many of the RPC-based systems of today.

The RPC mechanism allows the software developer to make a request to a server and wait for the response. An RPC call consists of the client sending a request to a known server, the server executing a specified procedure with parameters supplied by the client in the message,

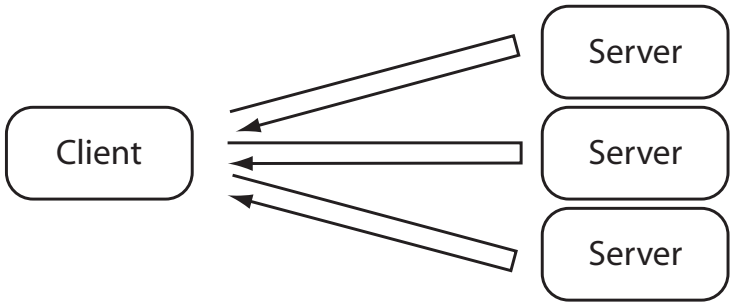

Figure 2.2: The use of synchronous messaging and the lack of an intermediary couples the client tightly to the server.

and finally the server sending a response back to the client containing a return value from the executed procedure.

From a software developer's point of view the RPC model is very attractive, because the provided abstraction resembles local function calls. The developer can in principle ignore the fact that the RPC call is passed over the network to a remote node. The developer is also able to move remote code back to the local process with relative ease by simply replacing the RPC calls with local function calls.

In practice though, while the RPC API hides most of the networking details of the RPC call, it cannot hide the increased latency, the possible network-related failure modes, or the fact that the referenced objects reside in separate memory spaces. Therefore the apparent simplicity of the RPC abstraction is often misleading, resulting in applications with, for example, insufficient error handling. It is not uncommon to find distributed applications based on RPC that fail to scale up in size, because of latent problems that have previously been masked by the small size of the deployed system [WWWK94].

The procedure call abstraction results in tight coupling between the client and the server in all the three dimensions described earlier (See Figure 2.2): (i) the client will block until the server responds to the request; (ii) the client must know the address of the server; and (iii) both parties must be running at the same time. Because of this tight coupling the RPC paradigm is not very well suited for large-scale, wide-area deployments where nodes are transient and link latencies are high.

Extensions to the RPC model have been introduced in an effort to try to relieve the tight coupling between the client and the server. For example, some implementations have introduced *fire-and-forget* style RPCs where the client does not care about the successful execution of the call nor the possible return value. With a fire-and-forget call the client is able to resume its execution immediately after making the call without having to wait for a return value.

Another popular extension to the RPC model, first proposed by Liskov and Shira in [LS88], are *futures*. This extension allows an RPC API call to return a future object to the caller rather than the real return value. The future is returned immediately while the remote procedure

call is handled in the background by another thread of execution. The background thread will eventually place the real return value into the future object once the remote call returns. Futures are a simple way to allow the client to continue its execution while a remote procedure call is being serviced in the background. When the client is ready to deal with the return value, it can read it from the future object. If the client does not have anything else to do before handling the return value, the future object typically implements a blocking `get()` method that allows the client to wait for the return value. Otherwise the client can use the `poll()` method to check if the return value has arrived yet and perform some other work while waiting for it.

Both extensions aim to enable the application programmer to use asynchronous messaging towards the server when possible, thereby increasing system performance and scalability when the client is not interested in the return value of the call.

Most modern, object-oriented RPC implementations, e.g. CORBA and Java RMI (See below), typically include a *name service*. Instead of using the server's address to access an object directly, the client uses the name of a service to look up the location of the object implementing that service from a registry. The location of the object includes the address of the server hosting that object. The name service introduces a level of indirection between the client and the server. This indirection allows the nodes that implement the service to change as long as the name-to-object mapping is updated in the name service. The name service can also be used to introduce some level of fault-tolerance to the system by allowing multiple objects to register with the same name. The name service will then load balance lookups between all the objects registered with the same name. Another alternative is to return a list of objects to the client and let the client pick one to use. This approach allows the client to transparently switch from one object to another in case of failure, assuming that the remote object does not maintain any session state.

There have been numerous implementations of the RPC paradigm since RFC 707. One of the most notable implementations is Sun's RPC [Sun88], which is used as the transport for the *Network File System* (NFS) protocol [SCR+03]. Sun's RPC was renamed in 1995 by the IETF as the ONC RPC (Open Network Computing RPC) in [Sri95]. The Open Software Foundation has standardised another RPC implementation as a part of their *Distributed Computing Environment* (DCE) [Cha93]. The latest version of the DCE specification, version 1.2.2, was released in early 2005. Microsoft adopted version 1.1 of DCE RPC as the basis for the MSRPC mechanism that is used to implement the DCOM framework in the Windows operating system.

The *Common Object Request Broker Architecture* (CORBA) [Obj04a] and Java's *Remote Method Invocation* (RMI) [Sun94] both provide an object-oriented RPC abstraction that allows clients to make method calls on remote objects that have been looked up from a naming service.

The latest additions to the RPC family of distributed communication paradigms include the Web services *simple object access protocol*[1] (SOAP) [Wor07] and *XML-RPC* [Win99], a

[1] The original acronym has been dropped since version 1.2 of the standard and the protocol is known now simply as SOAP.

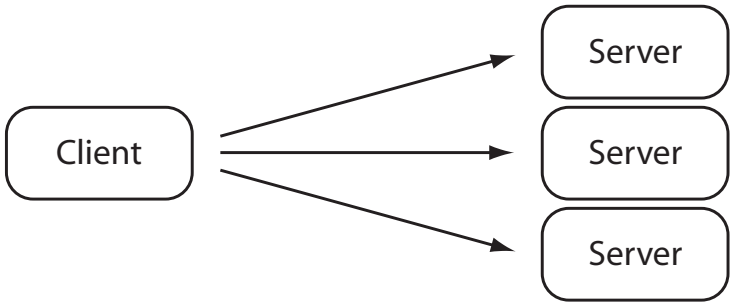

Figure 2.3: In traditional message passing the use of asynchronous messaging achieves synchronisation decoupling between the message producer and the message consumer.

simpler variation of SOAP. In both protocols RPC calls are serialised as XML messages and sent to a server usually as an HTTP request (although other transports like SMTP or XMPP are also supported).

2.1.3 Asynchronous Messaging

The alternative to synchronous RPC is asynchronous messaging. Instead of making a request to a server and waiting for a response, a node sends a message to the recipient. The message is one-way only, i.e. there is no response to it. If a response is required, the recipient will send a response as a separate first-class message.

Because the message sending does not result in a response from the recipient, it is simple to implement message sending in an asynchronous way and thus enable the client to carry on execution immediately after submitting the message to be sent. The underlying distributed middleware takes care of sending the message over the network to the recipient in the background (i.e. typically in another thread of execution).

Asynchronous messaging comes in a variety of flavours, of which publish/subscribe is one example. The other flavours include simple *message passing* and more advanced *message queueing*.

Message passing is one of the earlier forms of distributed interaction. In this paradigm an originator sends a message to a recipient, as shown in Figure 2.3. The sending of the message is asynchronous, i.e. the originator can carry on with its execution immediately. The receiving of messages on the other hand is typically synchronous, i.e. the recipient blocks on the message sink waiting for incoming messages. A simple implementation would consist of the originator sending UDP packets to the recipient that is blocking on reading a UDP socket.

With respect to the three dimensions of tight coupling discussed above, the nodes in a message passing system remain tightly coupled in space and in time: the originator must know the address of the recipient, and both nodes must be active at the same time.

Message queueing is another style of asynchronous messaging aimed at larger systems where the space coupling of message passing is not acceptable. In message queueing a *message*

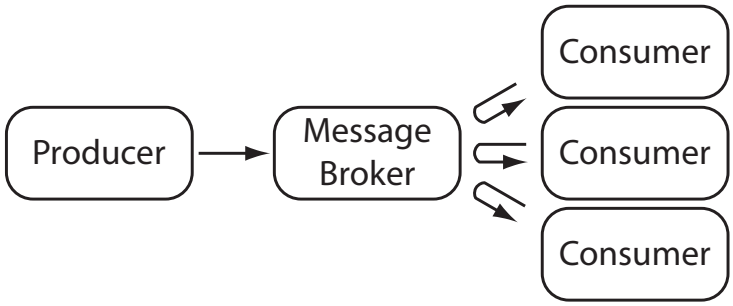

Figure 2.4: In a message queueing system the message broker decouples the producer from the consumers in time and space, but the consumers need to pull messages from the broker synchronously.

broker provides a level of indirection between the communicating nodes, not unlike the event service in a publish/subscribe system, which further decouples the nodes from each other (See Figure 2.4). Instead of the client sending a message directly to the server, the client sends the message to a *message queue* hosted by the message broker. The intended recipient of the message, i.e. the owner of the message queue, dequeues the message from the queue synchronously when it is ready to handle a new message.

Typically a message queue in the system can be read only by one consumer at a time. The same message cannot be read by multiple consumers. Therefore message queueing implements a *many-to-one* communication model, i.e. multiple producers can communicate with only one consumer over a message queue.

Compared to message passing the message broker decouples the message producer from the consumer both in space and time: the producer is required to know only the message broker rather than each consumer, and the message broker can store events for a consumer while it is not active. On the other hand the fact that consumers pull messages from the message broker creates a synchronisation coupling between the message broker and the message consumer.

One of the earlier *message oriented middleware* (MOM) implementations is IBM's WebSphere MQ [IBM07]. The *Java Messaging System* (JMS) [Sun02] has become a very popular messaging standard with the rise of Java's popularity in enterprise systems. Another, relatively recent implementation, is the Apache foundation's *Apache ActiveMQ* [Apa07] with its *OpenWire* wire protocol. ActiveMQ also implements a number of the more recent enterprise integration approaches like *Stomp* [Sto07], web services, and *REST* [Fie00] as well as JMS.

It is important to notice that asynchronous message passing is the underlying communication paradigm in most distributed system formalisms, e.g. Hoare's *Communicating Sequential Processes* (CSP) [Hoa78] and the *Pi-calculus* by Milner et al. [MPW92].

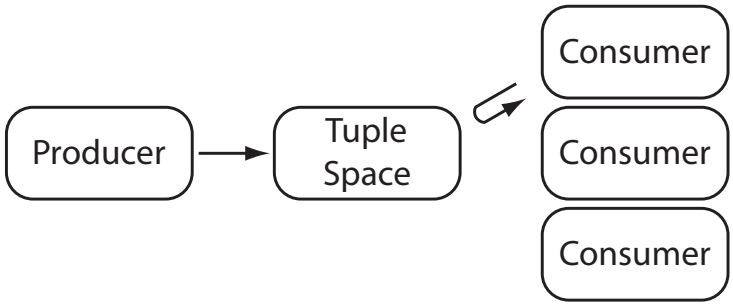

Figure 2.5: The *in* operation supports a many-to-one interaction model.

2.1.4 Tuple Spaces

A *tuple space* is an implementation of *distributed shared memory*. The shared memory, in this case, stores a collection of tuples. A node can access the tuple space by reading and removing tuples from the space and by inserting new tuples into the space according to the API shown in Table 2.2. Both the *read* and *in* operations accept patterns that are used to select a tuple from the tuple space. If more than one tuple match the pattern, one of the tuples is selected at random from the matching tuples and the operation is applied only to that tuple. All tuples are equally accessible to all nodes in the system, which allows the tuple space to be used as a communication medium between nodes.

Function	Description
in	consumes a tuple from the tuple space
read	reads a tuple from the tuple space
out	writes a tuple into the tuple space

Table 2.2: The tuple space API of three operations used to write, read, and consume tuples.

Depending on the operation used to read tuples, a tuple space can implement either *many-to-one* message delivery if the consumer consumes the tuple from the space, or *many-to-many* delivery if the message is read and left in the tuple space (See Figures 2.5 and 2.6).

The client nodes in a tuple space system are decoupled from each other, and from the tuple space both in time and space, but again, similarly to message queueing, the consumers pull messages from the tuple space synchronously.

Tuple spaces suffer from scalability issues. It is difficult to distribute the tuple space over a set of nodes. One approach, suggested by Xu and Liskov in [XL89], is to replicate the tuple space over a set of nodes in which case each node will contain all tuples. Murphy et al. in [MPR01] suggest another approach, targeted at mobile environments, where mobile nodes all have their own tuple spaces with their own content. A node will cache a tuple destined for an unreachable node in its own tuple space until the destination rejoins the system.

Neither approach is suitable for large-scale systems. In the former approach the tuple space

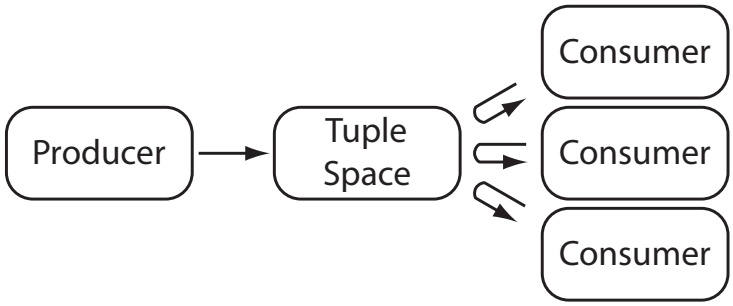

Figure 2.6: The *rd* operation in a Tuple Space allows many-to-many interaction between producers and consumers.

will grow in size indefinitely. In the latter approach all nodes in the system are expected to have knowledge of all other nodes in the system and the leaving or joining of a node causes the execution of global *engagement* and *disengagement* protocols.

The underlying problem is that it is hard to divide the tuple space into smaller sub-spaces that still allow for efficient pattern matching with the *in* and *rd* operations.

Some implementations, most notably JavaSpaces and Rinda, have extended the original interface with a `notify` operation that allows the tuple space to notify consumers of actions executed on the tuple space (e.g. adding or removing tuples). The notify operation decouples the consumers from the tuple space thus providing synchronisation decoupling.

Tuple spaces were first introduced as a part of the *Linda* programming language [Gel85]. Since then other implementations have emerged, e.g. IBM's *TSpaces* [WMLF98], Sun's *Java Spaces* [Sun03] specification and *Rinda*, a Ruby implementation of a tuple space [Rin07].

2.2 Publish/Subscribe Subscription Models

The publish/subscribe interaction model enables a publisher to publish an event to a set of subscribers. The term *to subscribe* implies that the subscriber is in control of what kind of events it receives. In order to empower the subscriber in this way, the publish/subscribe system must provide a way for the subscriber to express its interests in the form of a subscription. The subscription acts as a filter on the published events: the event service will deliver to the subscriber only those events that match the subscriber's subscription.

The subscription mechanism provided by the publish/subscribe system imposes a trade-off: a less expressive subscription mechanism results in many unnecessary events being delivered to the subscriber which are then discarded by the application; a more expressive subscription mechanism on the other hand requires the event service to do more work when delivering events [CRW99].

The following sections describe the following three publish/subscribe subscription models in more detail: *topic-based*, *content-based*, and *type-based* publish/subscribe.

2.2.1 Topic-Based Publish/Subscribe

The first publish/subscribe systems implemented a topic-based subscription model. In topic-based publish/subscribe published events are associated with *topics*. A subscriber, respectively, specifies a topic as part of the subscription. Instead of receiving all events published in the system, the subscriber is notified only of those events that have been published on that topic.

A topic forms a broadcast communication channel from all publishers to the subscribers of a given topic. Therefore, topics are very similar to *groups* in *group communication* systems [Pow96] where nodes join a group in order to communicate with other group members. In fact, one of the first systems to implement the publish/subscribe interaction model was built on top of the *Isis group communication toolkit* [BCJ+90].

However, the topic-based subscription model provides the subscriber only with very limited expressiveness. This results in the subscriber typically receiving many unnecessary events that have to be discarded in the application. For example, assuming that all stock quotes in the stock ticker example (See §1.2.1) are published on a single `StockQuote` topic, a subscriber, that is interested only in one of the many companies, will have to filter out all the other stock quotes itself. This results in inefficient use of resources as well as requiring the subscriber to do extra work.

A solution to this problem is to divide the topic space into a larger number of topics. Instead of having one topic to represent all stock quote events, we can publish each company's quotes under its own, company-specific topic. This approach allows subscribers to subscribe to company-specific topics and receive only those events that are interesting to them.

The downside of a verbose topic space is the fact that in order to subscribe to a larger segment of the topic space, the subscriber must subscribe to multiple topics. For example, continuing with the stock quotes example, a subscriber that wants to receive all stock quote events, regardless of the company, must now subscribe to all company-specific topics that number in the hundreds.

In order to address this expressiveness issue topic-based publish/subscribe implementations introduced *topic hierarchies*. Topic hierarchies allow the topic space in a publish/subscribe system to be organised into one or more hierarchies: topics related to each other are placed into a hierarchy according to some containment relationships. The sub-topics in a hierarchy represent a more specific sub-category of their parent topic. For example, a company-specific topic (e.g. `Nokia`) would be the sub-topic of a general `Stocks` topic, as shown in Figure 2.7. A subscription to a topic that has sub-topics results implicitly in subscriptions for all of the sub-topics as well. By representing the companies in a stock market and the rest of the stock exchange as topics in a topic hierarchy we can allow subscriptions both to company-specific topics and to the parent topic `Stocks`. In both cases the subscriptions are expressive enough to allow the event service to take care of all the event filtering without the subscriber having to do any application level filtering at all.

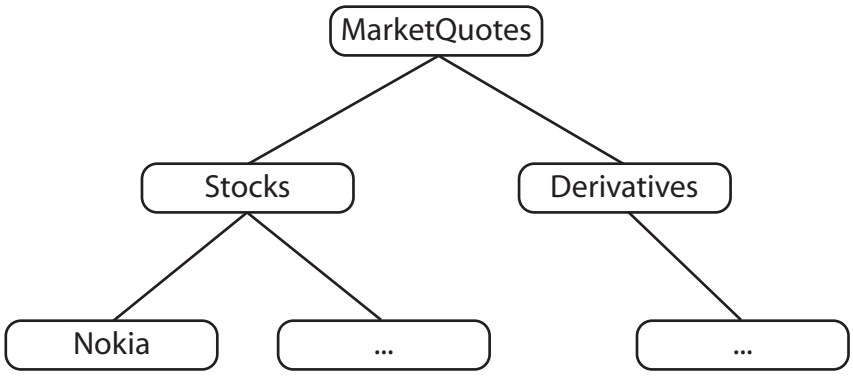

Figure 2.7: Subscribing to a topic in a topic hierarchy implies subscriptions to all sub-topics.

Another popular enhancement, which was first introduced by TIBCO Rendezvous [TIB07], is to support *wildcards* in topic names. Wildcards allow event clients to subscribe to, or publish under all topics in a set of topics matching the topic pattern. The wildcards in the TIBCO system allow the subscriber to replace a node in the hierarchy tree with a wildcard. For example, by subscribing to `MarketQuotes.*.Nokia` a subscriber is able to subscribe to all three-component topics in the system that start with `MarketQuotes` and end with `Nokia`. Assuming that the stock exchange example includes also company-specific derivatives, e.g. Nokia futures, this subscription will match Nokia-related events both under the `Stocks` and `Derivative` sub-topics.

Topic-based publish/subscribe has been implemented by a number of publish/subscribe middleware products including Altherr et al. [AEM99] and TIBCO [TIB07]. The JMS specification also defines some topic-based functionality.

2.2.2 Content-Based Publish/Subscribe

While hierarchical topics allow subscribers to describe their subscription in more detail, in many cases the expressiveness provided by topic-based publish/subscribe is simply not enough, resulting in either fragmented topic spaces or alternatively excessive application level event filtering and inefficient resource usage. The fundamental limitation in the topic-based subscription model is the fact that subscriptions are static, i.e. that matching of events to subscriptions does not take into account the content of the events.

The content-based subscription model, first proposed by Bacon et al. in [BBHM95], addresses this problem by allowing the subscriber to include an event filter expression in the subscription. The filter expression is applied by the event service to the content of each published event to determine whether the event matches the subscription or not. For example, a subscriber can subscribe to all events where `name = Nokia` and `price > 24.00`, which in topic-based publish/subscribe, event with topic hierarchies, results in filtering in the application.

Topic-based publish/subscribe can be seen as a special case of content-based subscription model where the subscription includes an equality filter over the `topic` attribute. Topic hierarchies can also be accommodated for if the subscription filters support string prefix matching. The content of the event would be another attribute that cannot be used in filters.

A subscription filter is specified in a *subscription language*, which allows a subscriber to define constraints on event content. Proposed subscription languages include SQL [Sun02], OMG's Default Filter Constraint Language [Obj04b], XPath [AF00, DFFT02], and publish/subscribe system specific proprietary implementations [BCM+99a, CRW01]. Nevertheless, most content-based publish/subscribe implementations implement relatively inexpressive subscription languages that support only conjunctions of simple comparisons (i.e. $=$, $!=$, $<$, $>$, $<=$, and $>=$). Some subscription languages, XPath and SQL, in particular allow for very fine grained filters including filtering on partial string content as well as dynamic filters based on, for example, mathematical expressions.

The choice of subscription language usually defines the set of data types available in publications. For example, simple comparison based languages typically support only basic data types: strings, integers, and floats. XPath 2.0 [Wor05] on the other hand supports up to 19 different data types including dates, timestamps, years, months, and URIs.

The challenge in content-based publish/subscribe systems is to implement event filtering in an efficient manner. Naïvely applying each subscription filter to each publication in turn will result in linear computational complexity with respect to the number of active subscriptions (i.e. $O(n)$). A more efficient alternative is to place the subscription filters into a tree where the most generic subscription is at the root and the leaves represent the most specific subscriptions. This approach allows the matching algorithm to ignore whole branches of the tree when a node in the tree does not match the publication resulting in logarithmic complexity (i.e. $O(\log n)$).

To be able to take advantage of this performance optimisation the event service must be able to impose a partial ordering on all subscriptions. We call this partial ordering the *coverage relation* between two subscriptions. Following the notation introduced by Carzaniga and Wolf in [CW01], we represent an attribute as a 3-tuple $\alpha = (type_\alpha, name_\alpha, value_\alpha)$ and a constraint as a 4-tuple $\phi = (type_\phi, name_\phi, operator_\phi, value_\phi)$. The constraint ϕ covers the attribute α, i.e. $\alpha \sqsubset_f^p \phi$, if $type_\alpha = type_\phi \wedge name_\alpha = name_\phi \wedge operator_\phi(value_\alpha, value_\phi)$.

A subscription covers, i.e. matches, a publication if all of its filter constraints cover the attributes present in the publication:

$$p \sqsubset_S^P f \Leftrightarrow \forall \phi \in f : \exists \alpha \in p : \alpha \sqsubset_f^p \phi.$$

A subscription f_1 thereby covers ($\sqsubset_S^S$) another subscription f_2 if f_1 covers at least all the publications that f_2 covers:

$$f_2 \sqsubset_S^S f_1 \Leftrightarrow \forall p \in P : p \sqsubset_S^P f_2 \Rightarrow p \sqsubset_S^P f_1,$$

where P is the set of all possible publications.

The subscription language must lend itself to comparing and ordering two expressions of that language. The comparison is relatively simple in restricted subscription languages that consist only of conjunctions of simple comparisons, e.g. `name = Nokia` and `price > 24.00`, but ordering arbitrarily complex filters becomes quickly intractable. Therefore decentralised publish/subscribe systems that rely on the ordering of subscriptions in order to be able to scale well typically implement relatively simple subscription languages.

The expressiveness of the content-based subscription model depends solely on the expressiveness of the subscription language. On the other hand, the efficiency of the event filtering implemented in the event service relies on the subscription filters implementing a partial-order. These two requirements introduce a trade-off between expressiveness and efficiency.

Some decentralised publish/subscribe systems, e.g. *Hermes*, use the coverage relation when distributing subscription state across the decentralised event service. Effectively the nodes in the subscription tree are represented by event brokers and each event broker will forward the publication to the other event brokers only if the publication matches the subscriptions on those event brokers. We will discuss event routing in Hermes in more detail in §2.3.1.

2.2.3 Type-Based Publish/Subscribe

Topics in publish/subscribe are used to group common events together. For example, a topic like `StockQuote` is used to group together stock trades that are happening in the stock market. In most cases events published under a given topic also share the same structure, i.e. they have the same set of name-value pairs.

The content-based subscription model, on the other hand, allows the subscriber to filter events based on the events content. By replacing the event topic with an event type, one can combine both the topic-based and the content-based models: events are published as instances of a type, which guarantees a given set of attributes, and the subscriber can filter on the contents of those attributes. This *type-based* subscription model also allows for better integration with programming languages, because an event can be provided for the programmer as a first-class object of the programming language.

In [EGD01] Eugster et al. propose a type-based subscription model that borrows heavily from object-oriented programming. In the proposed scheme event types define both attributes and behaviour. The attributes are expected to be private and only accessible through accessor methods. In addition to accessor methods, the type can also provide other methods that can be used in filter expressions or to access the event content in some indirect way.

It is unclear what the advantage of this encapsulation is, because the subscriber will most likely want to access all the fields that are included in the event. One can argue that the encapsulation allows for tighter programming language integration where objects are serialised as publications, but one could just as well allow the client to access the object's member fields directly either by making the fields public or generating direct accessors that simply return the field value.

The *Cambridge Event Architecture* (CEA) was the first event-based system to utilise a type-based subscription model [BBHM95]. CEA was built on top of CORBA and it used the *interface definition language* (IDL) to define event types. Events were instances of a class. CEA also supported content-based filtering as was discussed in §2.2.2. The Hermes publish/subscribe middleware, which we will describe in more detail in §2.3.1, inherited its type-based approach from CEA and the *CORBA-based event architecture* (COBEA) [MB98].

2.3 Decentralised Publish/Subscribe

We mentioned earlier in §2.1.1 that a publish/subscribe system consists of event clients and an event service. Depending on the size of the system, the event service can be embedded in the application with the event clients. In slightly larger systems the event service can be a separate service running on the same node with the event clients. In a distributed setting the event service can be a separate node with client nodes connecting to it over the network.

All publications travel via the event service. In a large-scale system with thousands of event clients the event service quickly becomes a bottleneck affecting the performance of the whole system. The next logical step is to decentralise the event service amongst multiple *event brokers* and distribute the system load across those brokers. The event clients connect to a *local broker*. The local broker acts as a *trusted* proxy between the event client and the rest of the event service, forwarding messages from the client to the rest of the system and delivering publications from the system to the client. By *trusted* we mean that the client trusts the local broker to proxy events for it to and from the event service without changing them. This includes event decryption and encryption as we will describe in Chapter 7. We assume that the local broker is either part of the same domain, or it is owned by a service provider trusted by the client.

The challenge in a decentralised publish/subscribe system is how to distribute subscription state and the event matching algorithm across the event brokers. One approach is to replicate all state and broadcast all publications to all brokers. This approach is simple, but it also results in a lot of unnecessary traffic between the event brokers. Other more advanced approaches aim to distribute state across all the event brokers which is more complicated, but requires less resources and results in more efficient bandwidth use, thereby improving the overall scalability of the system. The downside in the distributed approach is that each event broker is a single point of failure in the system, because the loss of a broker means that some part of the system

state has been lost. An optimal approach is to distribute subscription state across the event brokers as much as possible while at the same time replicating some of that state so as to provide enough redundancy to be able to survive the loss of one or more event brokers.

A number of decentralised publish/subscribe implementations have been proposed in the literature. One of the earliest implementations is the *scalable Internet event notification architecture* (Siena) [Car98, CRW01]. Siena is a content-based publish/subscribe system that was specifically designed for Internet-wide deployments.

Another significant implementation is the *Gryphon* project at IBM Research [BCM+99a]. Gryphon is an industrial-strength, content-based publish/subscribe system that has now been integrated to IBM's WebSphere suite of enterprise messaging products. Gryphon is based on an *information flow graph* (IFG) model [BCM+99b] where an IFG specifies the flow of information from publishers to subscribers.

Other notable implementations of decentralised publish/subscribe systems include the *Java Event-Based Distribute Infrastructure* (JEDI) from Politecnico di Milano [CNF01] and *Rebeca* from the Darmstadt University of Technology [FMB01].

The decentralised publish/subscribe systems can be divided into two camps depending on whether the event broker network implements a static or dynamic topology. A static topology is defined at deployment time and cannot change during the lifetime of the system. A dynamic topology on the other hand is able to adapt to changing network conditions and joining and leaving event brokers. Therefore a dynamic topology network can heal itself after node and link failures by re-balancing itself. From the systems mentioned above, Siena and Gryphon implement static topologies, while both JEDI and Rebeca implement a dynamic topology.

In a large-scale, Internet-wide system it can be assumed that event brokers will join and leave the system, and that network faults will cause event brokers to lose connectivity temporarily. Therefore a dynamic topology will make the broker network significantly more resilient to transient faults and node churn in an Internet-wide deployment.

2.3.1 Hermes

Hermes [PB02, PB03, Pie04] is a content-based publish/subscribe middleware with strong event typing. It implements a decentralised event service in order to provide scalable event dissemination and fault tolerance in the presence of node and network failures.

A Hermes system consists of *event brokers* and *event clients*, the latter being *publishers* and/or *subscribers*. Event brokers form an event broker network that performs event propagation by means of a type- and content-based routing algorithm. Event clients publish and/or subscribe to events in the system. An event client connects to a *local broker*, which then becomes *publisher hosting* (PHB), *subscriber hosting* (SHB), or both (CHB). An event broker without connected clients is called an *intermediate broker* (IB).

A feature of Hermes, that this work relies on, is support for *event typing*: every publication in

Hermes is an instance of an *event type*. An event type defines a *type name* and a set of *attributes* that consist of an *attribute name* and an *attribute type*. Supported attribute types depend on the types supported by the language used to express subscription filters. In our implementation the subscription language supports basic Java types that can easily be compared, e.g. integers, strings, booleans, dates, and floats. We will refer to this subscription model as *type-based* publish/subscribe throughout the rest of this dissertation.

Another Hermes-specific addition to traditional content-based publish/subscribe is support for *event type hierarchies*. In Hermes event types can be organised into inheritance hierarchies, where an event type inherits all of the attributes defined and inherited by its super-type. In addition to making defining new types easier, type hierarchies enable a subscriber to subscribe to a super type in an event type hierarchy and receive notification of events of that specific type as well as all its subtypes. While some of our work is compatible with event type hierarchies, the proposed design would require more work in order to support them fully. Therefore we do not claim to support Hermes' type inheritance. In general we have tried to keep the design as widely compatible with Hermes' flavour of type-checked content-based publish/subscribe systems as possible without any loss of generality.

We have built our access control system on top of *Hermes*. We chose to build on top of Hermes, because it is a decentralised publish/subscribe system with a dynamic broker network topology. Both features place requirements on the access control architecture. Therefore by concentrating on Hermes we provide an access control architecture that is equally applicable to decentralised, dynamic topology systems as well as centralised systems and system with a static broker network.

While parts of our work are also applicable to content-based and topic-based systems, we will concentrate on the type-based subscription model throughout this dissertation. Where applicable we have included a section describing how a certain feature could be implemented in a topic-based publish/subscribe system.

Event Routing

The event service in Hermes is implemented as a network of interconnected event brokers. The event brokers form a peer-to-peer system with each other where events are routed by means of *consistent hashing* [KLL+97]. More specifically Hermes is implemented on top of the Pastry *distributed hashtable* (DHT) [RD01b].

In a consistent hashing system each node picks a random identity for itself from a large identity space. Typically a random identity is generated by hashing some node specific information, e.g. the node's IP address, which results in the identities being uniformly distributed across the identity space. MAIA uses the SHA-1 hash algorithm to generate a node identity. The identity space is 160 bits. Messages in the system are sent to an identity. The destination identity is generated by hashing some information related to the message, i.e. a key. The message is then

routed to the node with the identity that is numerically closest to the target identity. All nodes in the system that know the key are able to access it by hashing the key with the hash algorithm.

Distributed hashtables (DHTs) [ZKJ01, SMK+01, RFH+01, MM02] use consistent hashing to partition the identity space amongst the nodes in the system. The key associated with a value is hashed and that hash value is used as the destination for *insert* and *lookup* messages.

Hermes (and MAIA) uses consistent hashing to find a *rendezvous node* for an event type amongst all the event brokers in the system. The rendezvous node is the node with the identity that is numerically closest to the hash value of the event type name. The rendezvous node is used as a meeting point for advertisement and subscription messages. Advertisement messages from the publisher hosting brokers and subscription messages from subscriber hosting brokers for a given event type are all routed to the same rendezvous node. Each intermediate broker that the advertisement or subscription message is routed through sets up routing state for the event type. Publications from the publisher hosting brokers are then routed through the event broker network according to the created routing state. More specifically, a publication follows the forward-path of the advertisement from the publisher to the rendezvous node. At every node where the publication meets a matching subscription a copy of the publication is sent on the reverse-path of the subscription towards the subscriber. The subscription paths form a tree routed at the rendezvous node. At each branch of the tree, the publication is again copied and one instance is routed towards each branch of the tree.

Because the identity generation in a consistent hashing system is effectively uniformly random, it is impossible to control the route an event will take through the system. In a multi-domain environment this means that a domain-internal message might be routed via brokers in other domains before it gets to its destination inside the originating domain. This places some requirements on the access control architecture. Namely the broker network cannot be trusted not to read or not to change the event content even though we trust all event brokers to route events correctly (See §3.6 for a more detailed discussion on the threat assumptions we have made). This is addressed in Chapter 7 where we introduce event encryption as a mechanism for enforcing access control in the untrusted broker network.

Event routing, including fault-tolerance of the routing state, in Hermes is described in more detail in [PB02, PB03].

2.4 Access Control

Access control in computer systems is used to control the type of actions a user can perform on a resource. In formal terms *objects* represent the resources that are being protected by the system, *subjects* represent, for example, users or processes performing actions on an object, and *operations* represent all the actions that the subjects can perform on the objects.

In an operating system access control decisions are mediated by the *reference monitor*. The

Classification
Top Secret
Secret
Confidential
Unclassified

Table 2.3: Common classification labels in decreasing order of access.

reference monitor was introduced as a concept by James Anderson in his study on computer security [And72]. Anderson proposed that the access control related functionality should be contained in a single component in the operating system that would be small enough to be subject to thorough testing and analysis. Together with hardware, firmware and other software the reference monitor in a computer system forms the *trusted computing base* (TCB). The TCB is defined as the set of components that, if working correctly, will be enough to enforce the security policy in the system regardless of the behaviour of other components. In other words, if the any of the components that form the TCB malfunction or contain a bug, it might jeopardise the security properties of the system.

Traditionally access control has been divided into *mandatory access control* (MAC) and *discretionary access control* (DAC) models. In MAC-based systems the system sets the access control restrictions for objects based on security policy and the subject that applied an operation on the subject. In DAC-based systems access control restrictions on objects are left to the discretion of the subject owning the object. In the recent past *role-based access control* (RBAC) has emerged as a third alternative access control model, which is powerful enough to simulate both MAC and DAC systems.

2.4.1 Mandatory Access Control

Mandatory access control systems have their roots in the military and intelligence communities, which have based their access control on hierarchical classification levels as shown in Table 2.3. MAC systems can only ever protect the confidentiality or integrity of data, but never both.

In 1973 David Bell and Leonard LaPadula presented *multilevel security* (MLS) [BL73, LB73, Bel74, BL76]. The MLS model concentrates on the confidentiality of data. It prevents information from flowing *downwards* in the classification system, i.e. from a higher level of classification to a lower one.

A subject in an MLS system is allowed to access an object only if its classification is greater or equal to the classification of the object. For example, a user with *Secret* classification is able to read and write *Unclassified*, *Confidential*, and *Secret* documents, but not *Top Secret* documents.

Ken Biba proposed another model [Bib75] that concentrates solely on data integrity, ignoring confidentiality considerations. When protecting the confidentiality of information it is

important to prevent that information from flowing from high classification levels to lower classification levels. On the other hand, in a system concerned with the integrity of information we must prevent information from flowing upwards from lower classification levels to a higher one. Subjects must always *read up* and *write down*, i.e. read data from a higher classification level and write data to lower classification levels. These goals are contrary to the goals of a confidentiality protecting system.

Other formal security models include the Chinese Wall security policy and the Clark-Wilson model. The Chinese Wall model was developed by Brewer and Nash in 1989 [BN89]. The model has its roots in the investment banking industry where it is important to internally prevent conflicts of interest, e.g. between the trading and the commercial banking departments. The Clark-Wilson model is another model that concentrates on data integrity rather than confidentiality. The model was proposed by Clark and Wilson in their 1987 paper [CW87]. Curiously the Clark-Wilson model has its roots in the accounting industry and draws many of its ideas from book keeping.

While formal MAC models enable reasoning about the security of the system and provably prevent malware like viruses and trojan horses from leaking information from the system, in many cases the models end up being too rigid for practical deployments. Operations that should be simple, e.g. object creation and deletion, become overly complex and require compromises.

With respect to access control in a multi-domain publish/subscribe systems, a MAC system would not be applicable simply because no single domain is in control of all the participating nodes. Simply by passing an event from one domain to another the originating domain has leaked the content of the event and has lost control over it.

2.4.2 Discretionary Access Control

Most operating systems provide discretionary access control (DAC). In a DAC system the subjects themselves are responsible for defining access control policy for their own objects. Access control policy is at the subject's own *discretion*.

Because the subject is able to define access control policy for her own objects, she is able to grant other subjects access to her resources. This allows the subject to, for example, share files with other users of the system.

Access Control Matrix

Access rights in a DAC system can be described with an *access control matrix* first proposed by Lampson in [Lam74]. An access control matrix consists of rows representing subjects and columns representing objects. The cells in the matrix define the operations that the subject can perform on the given object, as shown in Table 2.4.

More formally, as shown in Equation (2.1), where $M_{so} \subseteq A$ represents the access operations that the subject, $s \in S$, can perform on an object $o \in O$. Here S is a set of subjects, O

	/etc/passwd	/home/alice	/bin/sh
Alice	read	read, write, execute	read, execute
Bob	read	read, execute	read, execute
root	read, write	read, execute	read, execute

Table 2.4: An access control matrix representing files in a Unix system.

a set of objects, and A a set of all the access operations that a subject can perform on an object.

$$M = (M_{so})_{s \in S, o \in O}, M_{so} \subseteq A \tag{2.1}$$

While the access control matrix is a good theoretical tool, it is rarely used as such in actual implementations. The matrix is likely to be sparse in systems with more than one user where objects accessed by the users of the system rarely overlap. For example, in a typical multi user Unix system the users have their own files in their own home directories. The only files that are commonly shared between users are the executables in the system. Therefore, access control implementations typically use either access control lists or capabilities to represent policy [Gon89].

Access Control Lists

By taking a column centric view of the access control matrix, each column of the matrix is translated to an *access control list* (ACL). ACLs are typically stored with the object that the column represents. The ACL contains entries for each subject defining the operations that the subject can execute on the given object, as seen in Table 2.5.

In ACL-based systems it is often difficult to see which objects are accessible to a given subject. This is rarely a problem, though, because usually it is more interesting to get the list of subjects that are allowed to access a given object. If it is necessary to find all objects accessible to a given subject, for example if the subject's employment at the organisation has been terminated and the subjects access rights need to be revoked, it is possible to simply check each object in the system.

In a multi-domain setting the centralised nature of ACLs introduces some problems. It is difficult to deploy an ACL in a multi-domain system while guaranteeing that all nodes have the most recent ACL. It is also difficult to name subjects in a multi-domain setting while guaranteeing that names are unique as well as meaningful throughout the system.

More importantly for multi-domain environments, it is difficult to delegate access rights to other subjects. The delegator would either have to update a global, object-specific ACL to include an entry granting the delegate access to the object, or alternatively ask the resource owner to update the ACL. In the first case the ACL would have to also include an entry allowing

	...	**/home/alice**	...
Alice	...	read, write, execute	...
Bob	...	read, execute	...
root	...	read, execute	...

Table 2.5: An access control list represents one column of the access control matrix.

the delegator to update the ACL. In a multi-domain system cross-domain ACL updates would be cumbersome to implement. The ACL would also quickly become a performance bottleneck both when verifying or updating access rights.

For practical reasons access control lists are often truncated when they are implemented in operating systems. For example, in most Unix systems the ACL associated with a file contains only three subjects: *user*, *group*, and *others*. The motivation for simplifying ACLs in operating systems is twofold. First, as stated above, most files in a Unix system are accessed only by a few subjects or alternatively by a group of subjects, resulting in very sparse ACLs. Second, complete ACLs would need to be updated whenever a new subject is added to a system resulting in the management software having to go through all the ACLs of all the files in the system.

Capabilities

An access control system can alternatively be implemented with a row centric view of the access control matrix in which case each row of the matrix is translated to a *capability*. A capability contains entries for the objects that a given subject has access to, as can be seen in Table 2.6. While ACLs are usually stored with the objects, capabilities are often stored with the subject.

Because capabilities are often stored with the subject and the possession of a capability implies authority, it is important that a capability implementation protects the integrity of the capabilities. More specifically, capabilities must be unforgeable and non-transferable. That is, principals must not be able to forge capabilities and they must not be able to use a capability issued to some other principal. In centralised capability implementations capabilities are often protected by the hardware and the operating system. In a distributed implementation capabilities are usually protected by digital signatures to prevent forgeries, and they include the subjects identity, which is also protected by the signature, to bind the capability to the subject.

The Cambridge CAP computer developed in the 1970s is an example of a centralised capability-based system [Wil79]. The operating system of the CAP computer used capabilities in controlling access to objects. Instead of storing the capabilities with the subjects, the CAP operating system kept track of which capabilities each subject held internally. When making access control decisions the reference monitor would check that the capabilities held by the subject authorised it for the requested operation.

Capabilities have been making something of a comeback recently in the form of digital certificates. Especially in distributed systems it is desirable to give the capabilities to the subject

	/etc/passwd	/home/alice	/bin/sh
...	...	...	...
Bob	read	read, execute	read. execute
...	...	...	...

Table 2.6: A capability represents a group of cells on one row of the access control matrix.

to manage, i.e. it is the subject's responsibility to keep them safe and to present them when necessary. Capabilities in the form of signature protected certificates allow the issuer to give the capability to the subject without risk of tampering.

Capabilities support a very simple and elegant delegation mechanism: in addition to direct access rights for a given object, the capability can also state that the subject of the capability is allowed to further delegate those access rights. Again, because the capability's integrity is protected by the digital signature, there is no risk of subjects forging delegation rights. When delegating a capability the *delegator* creates a new capability for the *delegate*. When accessing the object, the delegate must show the verifier both its own capability as well as the delegator's capability.

A decentralised delegation mechanism is crucial in a multi-domain environment so that access control policy management can be distributed amongst all the participating domains without having to rely on a centralised party.

One implementation of a modern certificate-based capability system is the *simple public key infrastructure* (SPKI). Our multi-domain access control system will leverage SPKI-style capabilities. We will discuss SPKI in more detail in §2.6.

2.4.3 Role-Based Access Control

In both MAC and DAC systems, new subjects need to be added to and removed from the access control policy frequently. In a MAC system this is relatively simple, the new user is given a security classification that then grants her access to certain objects. In a DAC system, depending on whether its based on ACLs or capabilities, the new subject must be either added to all relevant ACLs or she must be issued capabilities to all relevant objects.

In many cases new subjects can be seen as new members of an existing role. For example, in a university a new lecturer will be a member of the lecturer role. *Role-based access control* (RBAC) [FK92, SCFY96] tries to leverage this fact in order to simplify access control policy management. Common groups of subject types are separated into roles that are granted access rights. Subjects are then issued with memberships to certain roles. The indirection introduced by the RBAC model allows new subjects to be added to the system with ease simply by granting them membership to certain roles that match the user's role in the organisation. There is typically no need to change the access rights of roles or subjects directly.

With respect to the MAC and DAC access control models, RBAC is mostly orthogonal to

both models. More specifically, RBAC can be used to implement both discretionary [SM98] and mandatory [OSM00] access control systems.

In this dissertation we concentrate on presenting an access control mechanism for multi-domain publish/subscribe systems that is based on capabilities. In this setting RBAC is a useful approach to simplify policy management in domains, but it has no direct impact on the capability-based access control mechanism. That is, one can implement RBAC within MAIA, but it is not necessary. We will touch upon policy management and RBAC in Chapter 6.

2.5 Decentralised Trust Management

Blaze et al. argued in [BFL96] that there exists a *trust management problem* in decentralised systems. At the time of the work, in the latter part of the nineties, the Internet was growing in leaps and bounds. The paper argued that traditional ACL-based access control systems were not appropriate for Internet-wide, decentralised systems that crossed domain boundaries.

Traditional ACL-based systems authenticate principals based on their name. For example, in an operating system the principal has a user name, which represents the principal in the context of that OS instance. The user name also maps to the principal's physical identity, i.e. the system administrator knows the mapping between user names and people. The principal authenticates herself to the system with a password associated with that user name. Blaze et al. argued that in the context of a computer system, i.e. a single administrative domain, the principal's name is well known and the mapping from principal to user name is appropriate and simple. But in a global setting, with multiple administrative domains, the management of names becomes cumbersome. Does the *Bob* in domain A represent the same principal as the *Bob* in domain B?

The X.500 [ITU05a] series of standards try to solve this problem by binding globally unique *distinguished names* to principals with X.509 [ITU05b] identity certificates. The distinguished name is based on hierarchical domain relationships and is therefore globally unique. It specifies, for example, the country, state, and city of the organisation that the principal is a member of. It is assumed that by leveraging hierarchical relationships the organisations that are issuing distinguished names are able to keep the names unique within their own namespace. The identity certificate allows a principal to authenticate herself as the owner of the distinguished name by proving ownership of the public key bound to the certificate.

But authenticating a principal to a name does not say anything about the authority of that principal. In traditional access control systems the authority of the principal is then looked up from an ACL. The ACL contains entries for all names, specifying the access rights of that name. Similarly with X.509 certificates, the access control system would look up the principal's distinguished name from an ACL to determine the principal's access rights. In a capability-based access control system the capabilities would be bound to the principal's name, as is the case with the X.509 *attribute certificates*.

In the end the access control system needs to know the principal's authority when deciding whether to perform the principal's request or not. The name of the principal introduces an unnecessary indirection in the authorisation process where the principal is first linked to a name, which is then linked to that principal's authority:

$$Principal \rightarrow Name \rightarrow Authority$$

Blaze et al. and the SPKI working group have both argued for removing the indirection introduced by the principal-to-name mapping and to simply map access rights to the identity of a principal, i.e. the principal's public key.

Another problem with ACL-based systems is the lack of a simple delegation mechanism, as discussed in §2.4.2. In a distributed system delegation of access rights allows access right management to be decentralised across the system. This allows the system to avoid performance bottlenecks that might otherwise appear if access control was centralised to a few nodes.

Similarly, the system must be able to decentralise access control decision making. That is, the nodes responding to the client requests must be able to make access control decisions themselves without having to rely on a central access control server. Delegation in the form of signed capabilities facilitates this type of decentralised authorisation.

2.5.1 PolicyMaker

Blaze et al. proposed the PolicyMaker *trust management engine* in [BFL96] as a solution to the trust management problem in decentralised environments.

To address the problems caused by the artificial indirection introduced by principal names, PolicyMaker uses public keys to represent principals. So the following bindings created by an X.509 identity certificate:

$$PublicKey \rightarrow Name \rightarrow Authority$$

would be replaced in PolicyMaker by the following binding:

$$PublicKey \rightarrow Authority$$

By treating the principal's public key as the principal's identity PolicyMaker avoids the complication introduced by the use of artificial names. Moreover, the public key is a globally unique identifier.

In the PolicyMaker model principals delegate access rights to other principals by issuing

signed capabilities that are called *assertions* in the PolicyMaker terminology. Assertions come with all the benefits of capabilities discussed in §2.4.2. The main benefit in a decentralised system is to allow principals to carry their own credentials rather than having to store them centrally in the system, as well as supporting access right delegation in a decentralised manner.

PolicyMaker can be implemented either as a dynamic library that is linked to applications, or alternatively as a system wide daemon process that can be used by all applications in the system. One of the key design goals in PolicyMaker was to separate the trust computation from the application logic and from credential management. As a result PolicyMaker expects the application to verify the validity of presented credentials, e.g. digital signatures and certificate validity dates. This approach has the added benefit that PolicyMaker is certificate and public key algorithm agnostic, which allows it to work both with current and future standards (e.g. X.509, PGP, RSA, DSA).

PolicyMaker takes as input a *request* to determine whether a public key (or a sequence of public keys) is authorised to perform a given action:

$$key_1, key_2, \ldots, key_n \text{ requests } action$$

The semantics of the action are application specific and they are not known to or interpreted by PolicyMaker.

In the PolicyMaker model principals are authorised to perform certain actions by an *assertion* of the following form:

$$source \text{ asserts } authority\_struct \text{ where } filter$$

The *source* is the source of the assertion. It can be either the local policy or the public key that signed a related capability. The *authority_struct* specifies the delegate of the assertion, i.e. a set of public keys that the assertion rule applies to. In most cases there is only one key mentioned in the rule, but the authority structure allows also for more complicated scenarios like threshold subjects, as described in §2.6.4. Finally, the *filter* is a predicate that is defined in a safe programming language. The subject's access request must satisfy the filter predicate in order to for the access request to be authorised by the assertion. In essence an assertion states that the source of the assertion trusts the public keys mentioned in the authorisation structure to perform the actions that are accepted by the filter program.

Other trust management engines in the vein of PolicyMaker include KeyNote [BFK98] and REFEREE [CFL+97]. KeyNote has been accepted as an IETF standard [BFIK99a, BIK00]. It was designed as a simpler descendant of PolicyMaker by many of the same people. REFEREE was originally designed to control access to web pages, nevertheless the trust management

engine is general purpose and can be deployed in other environments as well.

2.6 Simple Public Key Infrastructure

The *simple public key infrastructure* (SPKI) is another decentralised trust management system[2] with similar design goals than those of PolicyMaker et al. SPKI was designed by an IETF working group led by Carl Ellison. During the standardisation process the *simple distributed security infrastructure* (SDSI) [RL96] proposed by Rivest and Lampson was integrated into the SPKI proposal. SPKI was eventually standardised by the IETF in 1999 [Ell99, EFL+99].

The central idea in decentralised trust management and SPKI is to decentralise access control policy management, decision making, and credential management. This is achieved by implementing a capability-based approach to access control where the *owner* of an *object* is responsible for access control policy and credential management for that particular object. Distributing management responsibilities over all of the principals results in an extremely scalable access control system, because both management and verification can be implemented in a decentralised fashion without relying on centralised services that might turn into performance bottlenecks.

2.6.1 Authorisation Certificates

The main concept in SPKI is the *authorisation certificate*. An authorisation certificate is basically a signed capability. The certificate is represented by a 5-tuple: (I, S, D, V, A). *Issuer* is the principal that issued the certificate; *Subject* is the principal that the certificate is issued to; *Delegation* is a boolean value specifying whether the *Subject* is permitted to further propagate the *Authorisation* granted by this certificate; *Authorisation* is an application specific representation of the access rights granted to the *Subject* by this certificate; and *Validity* defines the date range when the certificate is valid and an optional set of on-line validity tests, e.g. *certificate revocation lists* (See Chapter 6 for a more detailed discussion on certificate validity and revocation). The *Issuer* field is either the issuer's public key or its hash value. The *Subject* field can be a public key of a principal, its hash, a name, the hash of an object, or a so called *threshold subject*.

An issuer can grant a subject a given set of access rights by issuing the subject an authorisation certificate that specifies those access rights in its authority field. The issuer can allow the subject to further delegate these access rights to other subjects by setting the *Delegation* field to `true`. This *certificate chain* can be infinitely long in SPKI as long as each certificate in the chain, except the last one, has the *Delegation* field set to `true`.

[2] According to Blaze et al. SPKI is, strictly speaking, not a trust management engine, because certificates can be processed in an application dependent manner, whereas a trust management engine, like PolicyMaker, processes all credentials the same way [BFIK99b].

To verify the access rights of a principal the verifier first verifies each certificate in the certificate chain independently. This includes verifying the digital signature on the certificate and all the possible validity conditions, e.g. the validity dates and the possible on-line checks. After the certificates have been deemed valid, the verifier must collapse a certificate chain into a single 5-tuple. The verifier reduces two adjacent certificates into a single 5-tuple with the following 5-tuple reduction rule:

$$\begin{aligned}(I_1, S_1, D_1, A_1, V_1) + (I_2, S_2, D_2, A_2, V_2) \\ \Rightarrow (I_1, S_2, D_2, A_1 \cap A_2, V_1 \cap V_2) \\ \text{iff} A_1 \cap A_2 \neq \emptyset \wedge V_1 \cap V_2 \neq \emptyset, S_1 = I_2 \wedge D_1 = true\end{aligned}$$

That is, for the reduction rule to be applicable, the issuer of the second certificate must be the subject of the first certificate and the delegation field in the first certificate must be set to `true`. The validity period of the resulting 5-tuple is the intersection of the validity periods of the two certificates. Similarly the authority implied by the resulting 5-tuple is the intersection of the two certificates.

The verifier applies the 5-tuple reduction rule to the certificate chain recursively until left with a single 5-tuple. The final 5-tuple is then used to determine whether the principal is authorised to make the given request. Notice that taking the intersection of both the authority and validity fields allows principals to issue authorisation certificates that have authority and validity fields that are greater than those of the previous certificates in the chain. Therefore if one certificate in the chain expires, that certificate can be refreshed independently of the other certificates. The system is therefore distributed both in space and time [Aur99].

The authorisation certificates communicate power from the issuer to the subject. This power is rooted at an ACL. ACLs in SPKI link a resource to a resource owner. When a verifier verifies an access request for an object, an ACL links the object to the issuer of the first certificate in the certificate chain, thereby forming a *certificate loop*. The SPKI RFC does not specify a format for ACL entries, however a sample implementation is simply an authorisation certificate without the issuer field.

A typical certificate loop is depicted in Figure 2.8 where the owner, P_a of an object O, grants P_b an SPKI authorisation certificate, C_{ab}, with access rights, $A_{ab} = (read, write)$, for the object O. P_b then further delegates access rights $A_{bc} = (read)$ to P_c by granting P_c another delegation certificate C_{bc}. Now, when P_c wants to access O, she shows P_a both certificates C_{ab} and C_{bc}. P_a is now able to form a certificate chain from P_c to P_b via C_{bc} and from P_b to itself via C_{ab}. Finally P_c authenticates herself to P_a by proving ownership of the key-pair P_c. P_c does this by executing a public key challenge-response protocol with P_a (See §5.3 for a more detailed discussion on challenge-response protocols). This completes the certificate chain which now, along with an implicit ACL, forms a *certificate loop* flowing from P_a to P_b to P_c and back to

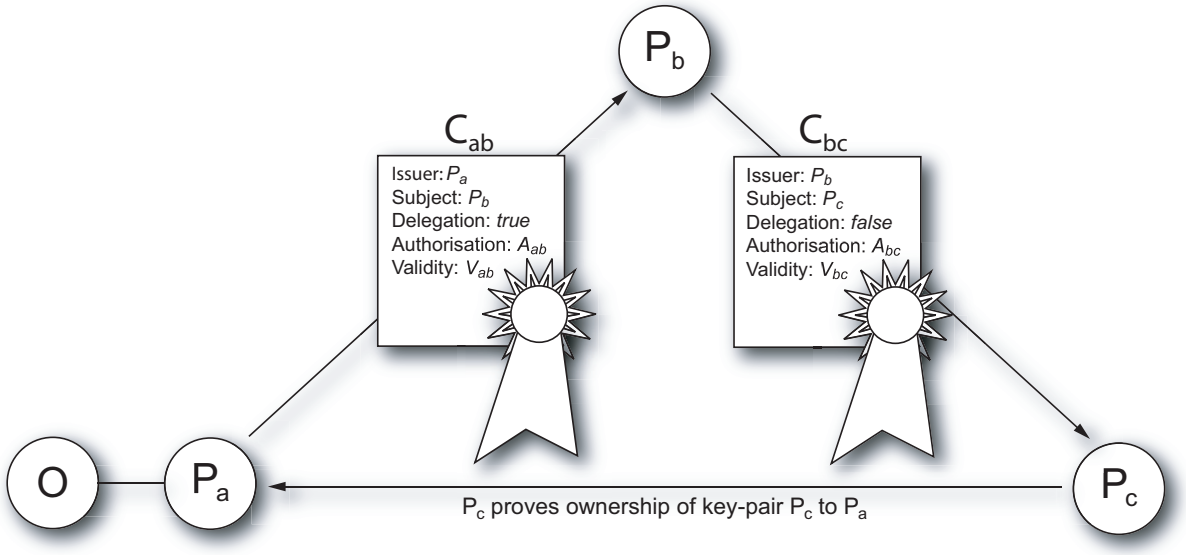

Figure 2.8: An SPKI authorisation certificate loop with three principals and two levels of delegation.

P_a again. P_a has now verified that P_c is authorised to access O within the privileges granted by $A_{ab} \cap A_{bc}$. Typically the verification is performed by an access control service rather than P_a.

Our work relies on SPKI authorisation certificates to propagate authorisation from resource owners to domains in a decentralised and scalable fashion. The following sections discuss some of the more advanced features of SPKI that are utilised in MAIA.

2.6.2 Name Certificates

One of the SDSI features that was incorporated into the SPKI specification as a result of the merging of the two was *name certificates*. In most cases SPKI relies simply on public keys as principal identities. But in some cases it is beneficial to be able to give that public key a name in some context. SDSI relies on linked local namespaces, i.e. each principal in the system has their own namespace and the principal is free to create whatever names it wants in that namespace. Another principal can refer to a local name in some other principal's namespace by prefixing the local name with the identity, i.e. public key, of the other principal. For example, if P_1 has defined a name foo to refer to P_2, a third principal can refer to P_2 with $P_1 foo$. The name can refer to either a principal or another name. Thereby we can create arbitrarily long linked names that will eventually refer to a principal with the principal's public key.

Naming is based on *name certificates*. A name certificate is a 4-tuple with the following fields: *Issuer*, *Name*, *Subject*, and *Validity*. The 4-tuple states that in the issuer's namespace the given name will refer to the given subject for the given validity period.

As mentioned earlier, when we introduced SPKI authorisation certificates, a name can be used as the subject in an authorisation certificate. That is, the authorisation certificate can dele-

gate authority to a name in the issuer's namespace. The issuer can then issue a name certificate for a principal, which links the principal to the authorisation certificate.

A name certificate will never be used in a trust calculation as such. It will always be reduced to a public key before being used in such a way. The name certificates allow also for a relatively painless integration of X.509 identities and SPKI. That is, the X.509 name can be mapped into an SPKI name certificate that is then used in the rest of the SPKI-based access control system.

2.6.3 Group Subjects

Another benefit of name certificates is the way they allow for the definition of *group subjects*. In the previous section we stated that an authorisation certificate can be issued to a name and a principal can be linked to that name with a name certificate thereby authorising the principal. The number of name certificates is not limited to one in SPKI, i.e. the issuer can issue any number of name certificates linking any number of principals to the same name. This effectively forms a group of principals that are all authorised by the same authorisation certificate. The mechanism can be used as a very simple RBAC system. We will use the idea of group subjects in §5.1.3 to grant a group of principals the same authority.

2.6.4 Threshold Subjects

SPKI supports also so called *threshold subjects* that allow multiple principals to act as a quorum. A threshold subject specifies k-of-n other subjects. Each subject wields $\frac{1}{k}$th of the power of the threshold subject. That is, at least k of the n subjects must delegate their shares for the authority of the threshold subject to be passed along. In practice each of the k subject issues a certificate independently of the other $k-1$ subjects. It is then up to the subject of those certificates to collect k certificates and show them to a verifier. The verifier will then verify each certificate and accept the statement made by the threshold subject if it has verified at least k separate certificates successfully.

Threshold subjects are desirable, because they incorporate redundancy. A combination of k principals is able to use the subject. Therefore, the threshold subject would remain usable even if some of the keys of some of the principals were compromised. Threshold subjects also support the removal of old principals and addition of new ones. Therefore a threshold subject can evolve through time and remain valid and secure for extensive periods of time. We propose the use of threshold subjects as owners for the resources that are present in MAIA in Chapters 3, 4, and 5.

2.7 Summary

In this chapter we have introduced the reader to the necessary background required to discuss both publish/subscribe systems as well as access control in the coming chapters. We have highlighted the features that allow the publish/subscribe paradigm to scale in wide-area environments and contrasted the publish/subscribe paradigm to the other distributed communications paradigms. We have also discussed the various types of publish/subscribe systems that exist and pointed out the features, e.g. type-based subscriptions and a decentralised event service, that we expect to see in multi-domain publish/subscribe systems. In the latter part of the chapter we described the basic theory of access control and moved on to describe decentralised trust management and SPKI that provide the foundations for our access control architecture.

In the next chapter we will present our definition for a multi-domain publish/subscribe system in an effort to provide scope for our work. We will also present a number of assumptions that we have made in our work so that we can build on these assumptions throughout this dissertation.

CHAPTER 3

Multi-Domain Publish/Subscribe Systems

In this chapter we will present our understanding of what a multi-domain publish/subscribe system is and how its users interact with it in an effort to scope our work. As mentioned briefly in the introduction in Chapter 1, we will consider only decentralised publish/subscribe systems that span across multiple independent administrative domains. We motivate our assumption that large-scale publish/subscribe systems will be implemented by multiple domains in §3.1. In §3.2 we present an informal definition of domains and describe each of the four types of domain members considered in our work. Access control systems are usually based on the notion of subjects or principals. We list in §3.3 the principals that are present in the MAIA architecture. We assume that a coordinating principal is responsible for forming the multi-domain system. We discuss coordinating principals in §3.4. We expect that the system will be deployed with encrypted connections between nodes in order to avoid a number of attack scenarios. We will discuss the details of encrypted connections in §3.5. In §3.6 we present an informal threat model in order to outline what security threats we are considering in this dissertation. Finally §3.7 elaborates on the numberplate monitoring example first presented in §1.2.2. We will use this application as a motivating example throughout this dissertation.

3.1 A Multi-Domain Publish/Subscribe System

Multi-domain publish/subscribe systems are large, distributed systems that extend across two or more independent administrative domains. A multi-domain system is formed when multiple domains connect their broker networks together, thus forming a shared event service consisting of brokers from all participating domains.

The incentive for domains to join the network is twofold: first, domains are interested in

implementing shared applications with other domains, e.g. publishers in one domain produce events while subscribers in other domains consume them. All domains that need access to a given application take part in forming the publish/subscribe system that this application is running on. Instead of setting up a separate publish/subscribe system for each separate application shared between two or more domains, the participating domains form a single large-scale publish/subscribe system that is used to implement a number of shared, distributed applications. Merging together all the small publish/subscribe systems used to run individual applications provides cost savings for all of the participating domains, because each domain can leverage the shared infrastructure rather than having to deploy a new infrastructure for each application.

Second, the shared infrastructure also provides the domains with a higher level of service in two ways: (i) a larger, shared broker network will generally be able to provide greater geographic reach without significant extra cost, and (ii) sharing a broker network will almost always increase the overall interconnectivity of the decentralised publish/subscribe infrastructure, thus providing a higher level of fault-tolerance and performance. We expect such a multi-domain system to consist of thousands of event clients and hundreds of event brokers, and to span a large geographic area.

In addition to multi-domain applications, we assume that domains will also want to deploy their private applications on the same shared publish/subscribe infrastructure. Again the incentives for doing so are the increased geographic reach, fault tolerance, and performance provided by the larger infrastructure, as well as the lower costs of not having to maintain two independent publish/subscribe systems, one private and the other public.

Shared infrastructures are attractive to domains, but only if the system provides appropriate access control mechanisms to prevent unauthorised access to deployed applications.

3.2 Domains

A domain in a multi-domain publish/subscribe system represents a physical or logical domain in the physical world, i.e. a corporation, an institute, or a department in either of the former. For example, a service provider, a stock exchange, a regional police force, or a university would all be considered domains. In principle a domain is an independent organisational entity that is responsible for maintaining its own publish/subscribe infrastructure.

The domains connect to each other either over private or public network connections. Private connections would be made over dedicated private network connections, or *virtual private networks* (VPNs). Connections over a public network would be over the public Internet or some other publicly accessible communication network that is not limited to the use of participating domains. In most cases we expect domains to communicate over the public Internet, but in some scenarios it is more likely that domains are connected either over dedicated connections or over virtual private connections. This would be the case in both the stock ticker example described

in §1.2 and the numberplate monitoring example described in §1.2.2 and in more detail at the end of this chapter in §3.7.

There are four kinds of components in a domain that are interesting to us: sub-domains, event clients, event brokers, and an access control service. We will discuss each domain component separately in the following four sections.

3.2.1 Sub-Domains

Domains can also be arranged in a hierarchical manner, which allows larger domains to organise their internal structure to smaller entities in order to facilitate easier management. For example, a large university might delegate publish/subscribe infrastructure management duties to individual departments instead of handling everything centrally. Similarly, an investment bank must for legal reasons separate its trading department from its corporate financing department in order to avoid the misuse of confidential client information.

The sub-domain hierarchy can be as deep as necessary, i.e. sub-domains can have sub-domains that have their own sub-domains etc. We use the concept of sub-domains to divide the enclosing domain into multiple trust domains. Each sub-domain is issued with its own set of access rights. Some of those access rights do not overlap with the other sub-domains thereby giving that sub-domain access to resources that are unavailable to the other sub-domains. A domain is allowed to contain either event brokers or sub-domains. Such a restriction allows for a very simple authorisation policy for event brokers where the event brokers inherit all the access rights of the enclosing domain. We discuss this in more detail in §5.1.4.

3.2.2 Event Brokers

Event brokers are the backbone of a decentralised publish/subscribe system, as explained in §2.3. The event brokers of a domain connect it to the shared, multi-domain publish/subscribe system. It is expected that each domain adds event brokers to the shared publish/subscribe infrastructure. The promise of added brokers acts as an incentive for existing domains to allow new domains to join the shared infrastructure.

The domain's event brokers join the multi-domain publish/subscribe system by connecting to existing brokers. As members of the broker network the brokers are expected to route events from publishers to subscribers even if neither the publisher nor any of the subscribers are a member of the broker's domain. That is, the broker acts as an intermediate broker on the event's path from the publisher to the subscribers. This is especially important if the event broker network is built on top of a peer-to-peer routing substrate, as is the case with MAIA, where event routing is based on dynamic routes that can change during the life-time of the system rather than static routes created by system administrators.

In addition to routing events as members of the broker network, the event brokers are responsible for exporting the publish/subscribe API to the event clients in their own domain. That

is, event clients connect to a local broker in order to access the event service, as explained in §2.3.

3.2.3 Event Clients

Event clients implement the applications that use the publish/subscribe system as a communications medium. An application that wants to publish or subscribe to events connects to an event broker as an event client. This allows the application to access the event service through the publish/subscribe API exported by the event broker.

We expect that an event client will always connect to a *local broker*, i.e. an event broker that is a member of the client's domain. This will allow the client to trust the event broker to handle confidential event content and in general act as the client's proxy towards the publish/subscribe system by, for example, forwarding the clients subscription, advertisement, and publication requests to the event service.

It is expected that all domains include both event clients and brokers. This follows from the requirement that all domains provide a set of event brokers to the shared infrastructure, and the fact that there is little motivation for a domain to join the multi-domain system unless it has clients that require access to one or more of the shared applications. As an exception to this rule one can envision domains that provide event brokers to a publish/subscribe system as an infrastructure service without having its own clients in the system. Another possible scenario would include service provider domains that provide their customers access to the multi-domain publish/subscribe system, but who do not have their own event clients otherwise (e.g. the brokerage firms in our stock ticker example in §1.2 provide their clients access to the publish/subscribe system). In such a case the customers can be seen as members of the service provider's domain.

3.2.4 Access Control Service

The last component in a domain is an *access control service* (ACS). The ACS is an abstract service that is responsible for managing and enforcing the access control policy of the domain. The ACS is described as an abstract service, because the concrete implementation of the ACS is not relevant in our work. Our work relies only on the assumption that the ACS issues SPKI certificates to domain members (i.e. sub-domains, event brokers, and event clients) based on the domain's access control policy and the credentials that have been granted to the domain by other principals in the publish/subscribe system.

We will use the terms *domain* and *ACS* interchangeably throughout the rest of this dissertation. For example, in Chapter 5 we write that a domain is granted a given access right. In practice this means that the ACS of that domain is granted the given access right. The ACS is responsible for delegating that access right within that domain according to the domain's access control policy.

3.3 Principals

Each of the domain members mentioned above including the domain (i.e. the domain's ACS) is a principal in MAIA. As in SPKI, a principal is identified by its globally unique public key. The principal authenticates itself to a verifier by proving ownership of the corresponding private key by executing a challenge-response protocol with the verifier (See §5.3).

Access rights are delegated to principals, who either further delegate them to other principals, e.g. domains delegate access right to their members, or use them to access resources or services, e.g. event clients and event brokers use the access rights to access the publish/subscribe system.

In many cases a principal is a human who executes an application that inherits the principal's identity for the duration of the session. The application might be provided with a set of credentials by the principal, which define the application's access rights. Or alternatively the application can inherit the principal's identity, which would allow the application to behave as the principal and activate credentials on demand, e.g. role activation in RBAC (we discuss RBAC in MAIA in more detail in §6.1). In the numberplate monitoring example described in §1.2.2 Detective Smith is a human principal.

The principal can also be a software agent. That is, a piece of software running on a specific node has been given its own identity and issued its own credentials. For example, we see the event brokers in a domain to be principals in their own right. The CCTV cameras in the numberplate monitoring example are examples of software agents as principals.

A subject in the system can have more than one key pair. Because each key pair defines an identity, a subject with multiple key pairs has effectively multiple identities in the system and is seen as multiple principals. Nothing in the key pairs or in the system links two identities that are used by the same subject to that subject or to each other. Access rights are principal-specific. Therefore, access rights issued to one principal cannot be used by the other identities of a given subject.

We do not use X.509 identity certificates in MAIA to bind the public key to a X.509 identity for two reasons. First, an X.509 identity is not necessary in the architecture – even with identity certificates the access control decision making and verification would be based on the key pair rather than the identity. Second, it is not clear what identity to give to certain types of principals. For example, software agents, like event brokers, have no clear identity that would be meaningful to humans. In some cases, e.g. for human principals, binding the public key to the identity of the principal is sensible, but this can, and should, be implemented outside of MAIA.

3.4 The Coordinating Principal

A multi-domain publish/subscribe system could be created, for example, at the initiative of one domain that creates an application that other domains want to access. The domain grants

other domains access to the application by inviting those domains to join the publish/subscribe system. Thus the multi-domain publish/subscribe system grows organically when more and more domains require access to applications implemented by members of the publish/subscribe system.

We call the domain that forms the publish/subscribe system the *coordinating principal*. We see the coordinating principal as the owner of the publish/subscribe system, i.e. the coordinating principal forms the multi-domain system and is responsible for deciding which other domains are allowed to access the shared infrastructure. In addition to controlling access to the publish/subscribe infrastructure, the coordinating principal also decides which domains are allowed to introduce new event types in the shared system.

The SPKI threshold subjects allow a k-of-n group of principals to behave as a single principal, as we discussed in §2.6.4. By creating a threshold subject a group of principals can act together as the coordinating principal and no one domain has control over the whole system. For example, three domains that want to form a shared publish/subscribe system together can setup a 2-of-3 threshold principal to be the coordinating principal. Now two out of three domains must agree on which other domains to invite to join the system, or which domains are allowed to install new event types.

The threshold subjects also support changes to the group of principals that form the threshold subject. For example, two of the three domains in the above example can decide to replace the third domain in the 2-of-3 threshold subject with a fourth domain. This ability of threshold subject's is very important from a management point of view, because it allows the system to evolve when domains join and leave the system or if a key has been compromised.

We would expect most multi-domain publish/subscribe systems to be created by two or more domains where all of the original domains would want to have a say in managing the system. Threshold subjects allow all of those domains to have an equal amount of control over the shared publish/subscribe system.

3.5 Transport Layer Security

We assume that all links (i.e. both client-to-broker connections and broker-to-broker connections) in the publish/subscribe system are protected by *Transport Layer Security* (TLS) [DA99]. Securing the communication links between nodes with TLS is a simple way to prevent trivial network sniffing attacks. Also, the encrypted transport guarantees that the application level messages have originated from the other peer instead of having been injected to the system by an adversary. This is especially important in publish/subscribe systems with dynamic event routing where forged routing messages can be used to corrupt the routing state of the system thereby bringing the whole routing network down.

TLS requires the authentication of at least one of the two end points for it to be secure against

a *man-in-the-middle attack*. The authentication is usually based on X.509 identity certificates. For example, in the secure version of the *hypertext transfer protocol* (HTTPS) the server is usually required to present an X.509 identity certificate that is bound to the server's hostname. If the server's hostname does not match the hostname on the certificate, the user is presented with a warning.

Using the server's hostname as an identity, it is possible in secure web traffic, because it is safe to assume that the server's hostname is registered in the *domain name service* (DNS). In MAIA it is not necessarily true that the broker's hostname has been added to the DNS or that it even has a static IP. Forcing the brokers to have static IP addresses and registered hostnames would prevent us from having mobile broker nodes. It would also add to the administrative burden of adding a new broker to the system, because the system administrator would have to add the broker's hostname to the DNS system and that hostname would have to match the identity in the broker's X.509 certificate.

In a typical HTTPS session the client connecting the server is never authenticated. This is partly because users very rarely have X.509 certificates issued to them, so they do not have a certificate to present to the server. Also, in most secure web applications the user authorisation is done at the application level once the TLS connection has already been established. For example, in a web banking application the user is queried for a login name and a password before she can access her bank account. If the user had a X.509 certificate issued to her, she would not have to login to the web bank, because the bank's web server would have already authenticated the user as part of the TLS handshake. Again in our case issuing X.509 certificates to event clients is not an option, because the event client might not have a clearly definable identity, as discussed in §3.3.

Instead of using X.509 identity certificates, we use SPKI authorisation certificates for authorising TLS connection end-points. We will discuss access control and principal authorisation in more detail in Chapter 5, but suffice to say at this point, we want to divide access control in the publish/subscribe system to network-level and application-level concerns. Network-level access means that a principal is authorised to connect to the publish/subscribe system. When initiating the TLS connection, both peers will present their network-level credentials to each other. By verifying those credentials both peers can be certain that they are connected to another authorised member of the publish/subscribe system. Once the TLS connection has been established the application-level access control can be implemented on top of the secure connection.

In the multi-domain publish/subscribe system the issuer of all network-level access rights is ultimately the coordinating principal. Therefore the verifier can check that the other peers certificate chain is rooted at the coordinating principal, and that the certificate chain authorises the peer to connect to this particular publish/subscribe system (See §5.2 for more details). Assuming that the connecting peer has been authorised, possibly indirectly, by the coordinating principal to connect to the given publish/subscribe system, the verifier should establish the TLS connection to the connecting peer. The connecting peer's identity is immaterial. The real ques-

tion is whether the peer is authorised to join the publish/subscribe system.

3.6 Threat Model

We present an informal threat model in order to clarify what kind of threats and attacks MAIA is designed to protect against and what kind of assumptions we have made concerning the environment where MAIA is deployed.

In our approach we have divided access control in multi-domain publish/subscribe systems into two levels: network-level access control and application-level access control. Network-level access control controls access to the publish/subscribe infrastructure, i.e. which event clients and brokers are authorised to connect to the broker network. Application-level access control, on the other hand, controls access to event types (we consider a set of event type definitions to be a publish/subscribe application) that are being published on the publish/subscribe system. That is, application-level access control policy defines which principals have the rights to subscribe to or publish events of a given event type.

Computer security is often described in terms of confidentiality, integrity, and availability of both data and services. In publish/subscribe systems this means that we want to protect the integrity and confidentiality of advertisements, subscriptions and publications, and the availability of the event service. By implementing access control we are able to address all three facets of security. By controlling access to the event service we can address availability requirements. By controlling access to publication and subscription rights we can address integrity and confidentiality requirements.

We place a lot of trust on authorised principals. Basically we assume that principals that have been authorised to join the publish/subscribe system are trustworthy and not malicious. For example, we assume that an event broker that has not been authorised to access events of a given type might be interested in reading those events and thereby try to circumvent the access control system, but it will always route the events correctly. On the other hand we assume that if a principal were to start behaving maliciously, i.e. flooding the network with messages, corrupting routing state, or tampering with event content, we can identify them and revoke their access rights. Obviously principals can behave maliciously in ways that are difficult to detect. For example, an event broker can randomly drop events, and all principals can leak confidential information from the system to unauthorised parties (this is true for all discretionary access control systems).

We can identify four types of adversaries for a multi-domain publish/subscribe system: (i) an external adversary that is not a member of any of the participating domains and is therefore only able to eavesdrop on the message traffic between domains; (ii) an internal adversary that is a member of one of the participating domains, but does not have access to the publish/subscribe system and can therefore only eavesdrop on the message traffic between event brokers and

clients; (iii) an internal adversary that is authorised to access the publish/subscribe system, but is not allowed to access a specific event type; and finally (iv) an internal adversary that has access both to the publish/subscribe infrastructure and limited access to a given event type (e.g. subscription rights, but no publishing rights).

The goal of an adversary is to exceed her authority, i.e. gain access to system in a way that is not authorised by the access control policy. For adversaries in cases (i) and (ii) this means accessing the publish/subscribe system in any way possible. For adversaries in cases (iii) and (iv) this means subscribing to events, publishing events, and setting and reading attribute values in published events.

We can easily protect against both adversaries (i) and (ii) by deploying the broker network on top of TLS, as described above in §3.5. TLS secured connections would be used both for intra and inter domain connections. This prevents the adversaries from accessing any of the publish/subscribe system messaging, thereby preventing them from reading published events and from injecting messages into the system (e.g. publications or routing messages).

In the last two scenarios, i.e. (iii) and (iv), the adversary has limited access to the publish/subscribe system and she wants to exceed her authority and access additional event types. For both scenarios we require a more sophisticated access control mechanism. In Chapter 5 we propose an access control system that allows us to delegate access rights to domains and event clients, and to verify those access rights at the event client's local broker. In Chapter 7 we propose an approach to enforce access control within the broker network by encrypting event content.

We do not address more advanced attacks in this dissertation, e.g. attacks based on traffic monitoring.

3.7 Example Application

The architecture presented in this dissertation is motivated by problems facing organisations with which we have done collaborative research, such as the National Health Service (NHS) – particularly electronic health record management – and the Police Information Technology Organisation (PITO) in the UK. In our example application we consider the British Police Force – a federation of more than fifty largely autonomous regional forces. Historically the national police forces have been relatively independent of central policies in their decision making. For example, the police forces have been free to purchase and deploy software and data models independently of each other. This has resulted in incompatible information technology deployments across the forces, which hinder effective police work by preventing one force to access data and services from another force.

Many of PITO's projects aim to increase the efficiency of communications between the independent police forces. This is a challenging task, given the diversity of software deployed,

and the different ontologies and data models used within the separate forces. The main efforts in PITO include developing a data model for nation-wide use that is compatible with the data models used both by Interpol and Europol, and enabling nation-wide access to force-specific databases and national registries (e.g. fingerprint registry, vehicle registry, and the criminal records database).

We feel that a shared, nation-wide publish/subscribe system would enable information to flow more efficiently and faster from force to force. For example, publish/subscribe messaging allows a user to be notified when an event has occurred. This functionality is very valuable in criminal investigations where new information is most valuable when it is first discovered. For example, vehicle sightings, arrests, and the recovery of stolen goods or firearms are all events that can help the police solve an investigation. A publish/subscribe system allows the user to be notified of an event as soon as it happens, asynchronously, without having to constantly poll the related databases.

By implementing a common, shared publish/subscribe system, the national forces are able to share these applications nation-wide. The forces also benefit from lower infrastructure costs and added fault-tolerance with respect to the event service, as discussed earlier in the beginning of this chapter.

Although the regional forces are all part of the national police force and thus trusted, there still needs to be access control in place to provide confidentiality and to guarantee message integrity. For example, investigations include witness statements where the witness' identity must remain confidential in order to protect her privacy. Thus an infrastructure shared among multiple seemingly mutually trusting domains must implement an access control system such as the one proposed in this dissertation.

In addition to protecting data confidentiality and integrity, access control is also necessary in order to implement domain-internal applications. In the case of the British Police Forces, the regional forces will still have their own proprietary applications that they will want to maintain and possibly integrate with the publish/subscribe system. These applications should not necessarily be accessible to all the other national forces, i.e. the local force must be able to restrict access to those applications to the local force even when they are deployed on the nation-wide publish/subscribe system.

The threat model for this scenario is different from the threat model presented in §3.6, i.e. we can assume that there are no external adversaries, because the system is deployed within the police network, and the internal adversaries are more likely to be curious rather than malicious. Nevertheless we will assume the threat model from §3.6 in this dissertation in order to provide an access control architecture that will be applicable to a wider range of deployments and applications.

We will use the numberplate monitoring application that we touched upon in §1.2.2 as an example throughout this dissertation. Figure 3.1 shows the multi-domain publish/subscribe system consisting of three particular sub-domains:

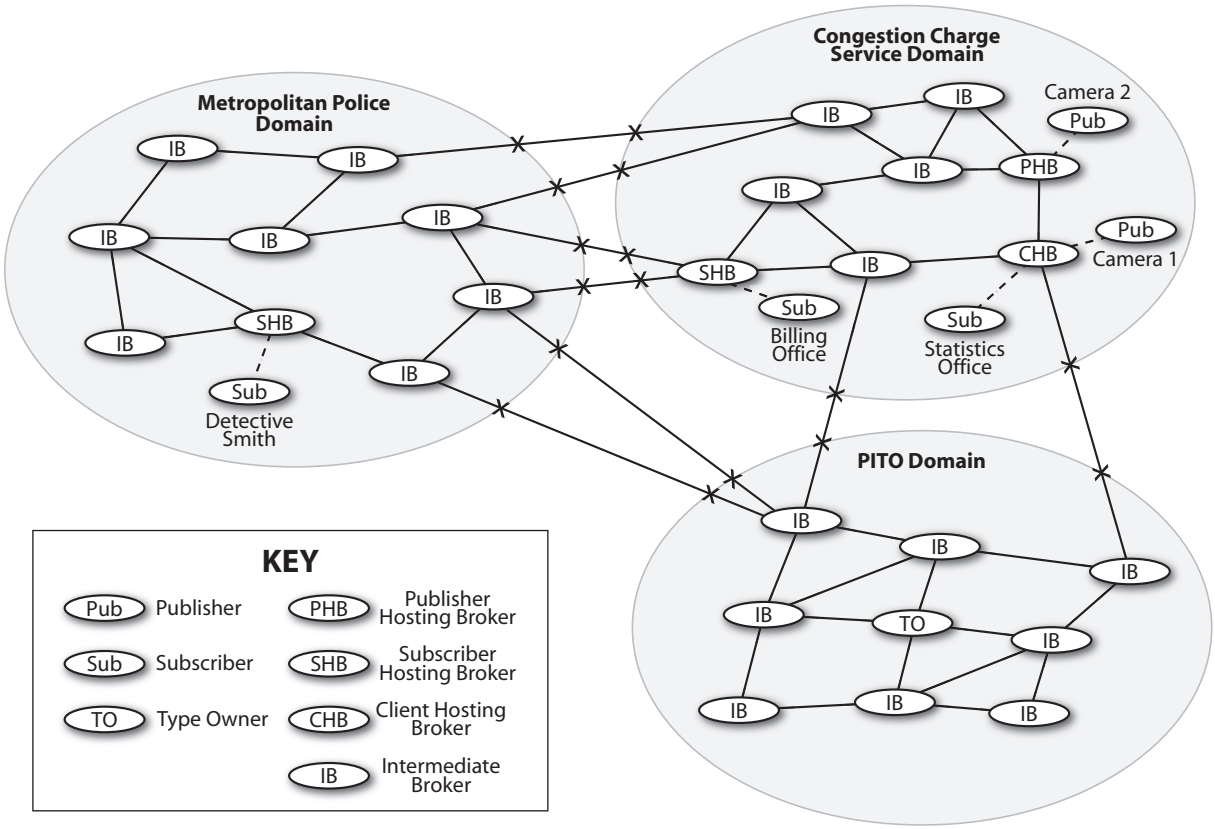

Figure 3.1: An overall view of our multi-domain publish/subscribe deployment

Congestion Charge Service Domain. The CCS domain implements the numberplate monitoring system in London. The main purpose of the system is to enforce payment of the congestion control charge by vehicles entering the congestion controlled area. The domain contains the CCTV cameras that implement numberplate recognition and publish numberplate events whenever a vehicle passes by them. The domain also includes the billing and statistics systems that levy the congestion control charges and monitor the number of vehicles that have passed through the London Congestion Charge zone each day. The fact that the subscribers in the CCS domain are only authorised to read a subset of the vehicle event data will exercise some of the key features of the enforceable publish/subscribe system access control presented in this dissertation.

Metropolitan Police Domain. The Met domain has access to the numberplate sighting events published by the CCTV cameras. The publications are used to track vehicles in London that are related to an ongoing investigation. The tracking is authorised on a case by case basis by a judge issuing an appropriate court order. The requirement of a court order is enforced by an access control policy in the Met domain as will be described in Chapter 6.

PITO Domain. The Police Information Technology Organisation (PITO) is the centre from which Police data standards are managed. It is the Coordinating Principal that formed

the publish/subscribe system and it has deployed the *Numberplate* event type that is used to report numberplate sightings in this particular scenario. PITO is an example of an infrastructure provider, i.e. it has no event clients of its own in our example scenario.

We also have the following four event clients in the example scenario deployed in the above three domains:

Detective Smith. Detective Smith is a member of the Met domain. She has been assigned to a case where the suspects were seen to have driven away in a car with the license plate `AE05 XYZ`. Detective Smith has been authorised to subscribe to numberplate events matching the suspect's numberplate by a court order for a limited time.

Billing Office. The CCS domain contains a Billing Office subscriber. This subscriber is responsible for receiving all *Numberplate* events and levying the vehicle owner the congestion charge whenever the vehicle is seen to enter the congestion controlled area. The Billing Office needs to see only the numberplate and timestamp information from each event to be able to levy the charge. In order to protect the vehicle owner's privacy the Billing Office is not authorised to read the location information.

Statistics Office. The Statistics Office is another subscriber in the CCS domain. Its purpose is to collect traffic statistics from the system. Similarly to the Billing Office, the Statistics Office requires only partial access to the *Numberplate* events. Specifically the Statistics Office needs to know the location and timestamp of the sighting, but not the numberplate of the vehicle that was sighted.

Cameras. Finally the system includes a number of cameras that publish *Numberplate* events. We assume that the cameras are able to execute the numberplate recognition software internally and publish the numberplate in textual form in a *Numberplate* event. Each camera knows its own location, which will be included in the published events alongside the time when the vehicle was sighted and the vehicle's numberplate. We expect that the CCTV cameras will be able to produce relatively accurate timestamps.

We will use this application as a running example throughout this dissertation. Many of the features of the MAIA system are motivated by the access control requirements of this example application. For example, the access control system must be able to enforce access to the events per event attribute, so that the system can appropriately protect the privacy of vehicle owners.

3.8 Summary

In this chapter we have outlined the scope of our work. We defined what a multi-domain publish/subscribe system is; how one is to be deployed; and what kind of threats it should be protected

against. Our aim has been to introduce the reader to the assumptions that we have made in our work, especially those assumptions that will affect some of the design choices that we make later on in this dissertation.

In the next chapter we will present a scheme for secure event type definitions, i.e. event type definitions whose integrity and authenticity can be verified by the user of the event type. The event type definitions will also provide globally unique and verifiable event type and attribute names. We feel that these features are the cornerstones of an access control system for publish/subscribe systems.

CHAPTER 4

Secure Event Types

The goal of our work is to design an access control architecture for decentralised, multi-domain, type-based publish/subscribe systems. We have designed our access control architecture for Hermes, i.e a publish/subscribe system with a decentralised, dynamic topology broker network, and a type-based subscription model, but the work is equally applicable to centralised systems or systems with a static broker network.

We lay the foundations for an access control architecture by first addressing the security issues of type definitions. For example, the lack of unique and verifiable names in Hermes-style event type definitions makes it very difficult to reference event types and attributes from an access control policy unambiguously.

In §4.1 we discuss event type definitions in general, introduce the basic Hermes type definition framework and address each of the security vulnerabilities inherent in those type definitions in turn.

We present *secure event type definitions* for Hermes-style type-based publish/subscribe systems in §4.2. Our model provides a cryptographically verifiable binding between type names and type definitions. It also facilitates self-certifiable type definitions that enable the verification of the authenticity and integrity of these type definitions. We argue that secure, unique, and verifiable type and attribute names are a prerequisite for comprehensive access control architectures in decentralised publish/subscribe systems. Although the chapter concentrates on type-based publish/subscribe, the naming scheme is also applicable to topic-based publish/subscribe. This will be elaborated on in §4.6.

In §4.3, we consider the management of event type definitions in multi-domain publish/subscribe systems and present a scheme for event type version management that supports multiple versions of an event type to be present in a publish/subscribe system at any given time.

Field	Description
name	Name of the event type definition
extends	An optional reference to an inherited event type
attributes	A set of attribute definitions

Table 4.1: A Hermes-style event type definition.

Following type versioning, we present a mechanism for the original type owner to delegate management duties to other principals by issuing them signed capabilities authorising them to manage a given event type definition.

We had to redesign some parts of Hermes in order to implement secure event types. However, the addition of secure event types allowed us to simplify other parts of Hermes. All in all the addition of secure event types makes the overall Hermes system simpler, as well as providing us with a solid foundation on which to build our access control architecture. These changes will be discussed in more detail in §4.4.

We evaluate our approach in §4.5 and discuss the computational overhead introduced by the addition of secure event types, or more specifically the cryptographic operations used to verify secure event types. We will show that all of the added overheads are one-time costs, i.e. the added overheads will affect the start-up time of brokers and the time required to process advertisement and subscription messages whereas event publication and delivery will not be affected.

Finally we conclude the chapter with a section covering related work, §4.7, and a section summarising the chapter, §4.8.

4.1 Event Type Definitions

Types in the context of publish/subscribe systems mean that published events must conform to a predetermined event type definition. We assume that a type is defined by a *type owner*, i.e. principal in the publish/subscribe system, who then provides the type definition to publish/subscribe applications and the event service. In most cases we would expect this principal to be either the domain, or alternatively a principal in a domain who is responsible for managing event types in that domain. The event service (i.e. local broker in the context of a decentralised publish/subscribe system) is responsible for type-checking a submitted publication against an event type definition provided in an earlier advertisement request.

A type definition can be modified after is has been first defined and re-deployed as a new version of the original type (See §4.3 for more details). We call a principal that modifies an existing type a *type manager*. Again, a type manager is expected to be a domain or a specific type manager principal in a domain. We refer to the type owner also as a type manager in the following sections when the type owner is modifying an existing type.

A Hermes event type definition, as shown in Table 4.1, consists of three components: an event type name, a set of attribute definitions, and an optional reference to an inherited event type (i.e. the name of the inherited type). An attribute definition defines the name and type of the attribute. A type definition name and an attribute name are both represented by a character string. As stated in §2.3.1, we will not consider event type inheritance in this dissertation. Therefore we will ignore the *extends* field of the Hermes event type definition throughout the rest of this dissertation.

The main problem with simple, Hermes-style, type definitions, as described above, is their lack of ownership information. The type definition does not specify who has defined it, i.e. its owner. In a decentralised access control system the resource owner is responsible for managing the access control policy for a resource. If the owner of an event type is not known, the system does not know whom to trust as the source of access control policy.

Related to the lack of ownership information, the type definition provides no integrity protection. Without integrity protection anyone will be able to change the type definition and claim that their version of the type definition is the authentic one. In a closed, small-scale system this may be acceptable, but in a decentralised, multi-domain publish/subscribe system this is not acceptable. In addition to malicious changes, a large-scale system is also more likely to experience, for example, transmission errors that might change the type definition accidentally. Both malicious and accidental changes to the type definitions would go unnoticed without integrity checks embedded in the type definition.

Finally, event type and attributes names are not protected against name collisions. For example, two type owners might both introduce an event type named *Location* at the same time without knowing about each other. In such a case it would be impossible for the access control system to know which event type definition was meant when it encounters the name *Location* in an access control policy or a publish/subscribe request. In addition to name collisions, the authenticity of type names cannot be verified. Authenticity in this case means that the event type name was defined by the same principal that defined the event type, i.e. the type owner. Without a mechanism for verifying the authenticity of types names a malicious principal could bind an existing type definition to a new name owned by him, which might allow the malicious principal to circumvent the access control policy by accessing the type definition through the new name.

The possibility of name collisions and the lack of a mechanism for verifying the authenticity of type names means that one can not reference type names or attribute names from an access control policy, because there are no guarantees that name in the policy references the correct event type. That is, there exists a many-to-many relationship between names and event type definitions instead of the expected one-to-one relationship.

The next section describes simple enhancements to the basic event type definitions that result in verifiable type definitions, and unique and verifiable names, with no possibility of name collisions.

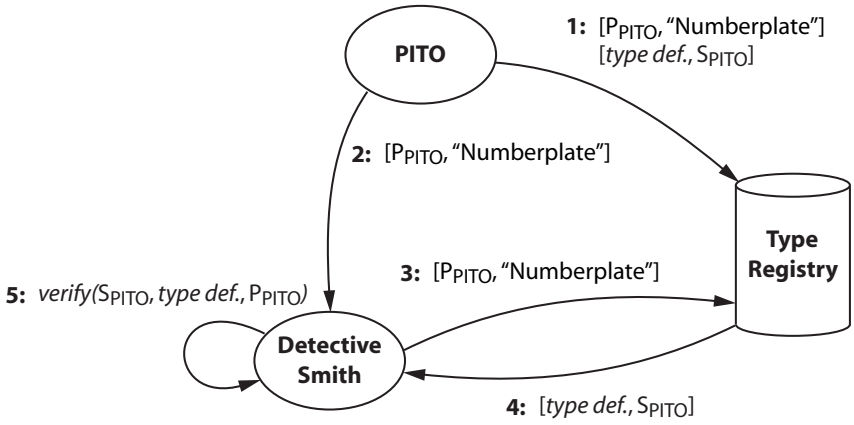

Figure 4.1: Detective Smith retrieves the Numberplate event type definition from a type registry and verifies its authenticity and integrity.

4.2 Secure Event Types

We propose a secure event type framework for type-based publish/subscribe systems that securely binds the type name and type definition to each other. It also guarantees type authenticity and integrity by using public key cryptography. Secure event types form a basis for our access control architecture by allowing types and attributes to be referenced from an access control policy in a secure and unambiguous fashion.

We approach the problem of secure event type definitions by defining a secure namespace for type names. We propose incorporating the type issuer's *identity* to the type name in the form of a public key. A public key is globally unique if the public key scheme is secure [EFL+99], thus a public key can be used to define a globally unique namespace. Namespaces that are specific to type owners will prevent both accidental and malicious name collisions in the publish/subscribe system.

Finally we propose that the type owner digitally signs the type definition. Because the private key used to sign the type definition corresponds to the type owner's identity incorporated in the name of the type definition, the type name is bound to the type definition by the digital signature. Similarly to digitally signed email messages, both the authenticity and integrity of the type definition can be verified by verifying the digital signature on the type definition with the identity in the type name. If the signature verifies correctly, it follows that the issuer is the owner of the namespace defined by the public key (authenticity), and that the type definition has not been tampered with since it was issued (integrity). Only the owner of the namespace is able to issue new types for that namespace, because the digital signature on the type definition must be bound to the identity defining the namespace. By incorporating the type owner's identity in the type definition's name we create a self-certifying link between the name and the type definition. That is, any principal in the system is able to verify the authenticity and integrity of any given event type definition simply by verifying the digital signature on that type definition

Field	Description
name	A tuple consisting of the type owner's identity and a human-readable name
attributes	A set of attribute definitions
signature	The type owner's digital signature containing all of the above elements

Table 4.2: A secure event type definition.

with the public key that is part of the type definition's name. Figure 4.1 shows how Detective Smith looks up the *Numberplate* event type definition from the type registry and then verifies the authenticity and integrity of the type definition by verifying its digital signature with the public key present in the name tuple. The figure presents the following steps:

i. The type owner, PITO, inserts the *Numberplate* event type definition into the type registry.

ii. PITO sends the name tuple to Detective Smith,

iii. Detective Smith looks up the event type definition matching the name tuple she received from PITO,

iv. The type registry returns the event type definition with its digital signature, and finally

v. Detective Smith verifies the digital signature S_{PITO} with the public key P_{PITO} that is part of the name tuple.

At this point a secure event type definition consists of three items as depicted in Table 4.2. We will add two more items to the secure type definitions in §4.3 to facilitate type management.

The following sections discuss first the changes to the name of the type definition and then the digital signature.

4.2.1 Name Tuple

The name of an event type is used to refer to a type definition from publish/subscribe messages (i.e. advertisement, subscriptions, and publications). The name can also be used, for example, to request a type definition from a type repository. It is crucial that a specific *type name* references the same *type definition* for all clients and messages, i.e. the name tuple must identify a type definition unambiguously.

We replace the traditional type name with a 2-tuple consisting of the *type owner's identity* as a public key and the traditional *human-readable name* (we will add a third item to the name tuple, a version number, in §4.3.1). The name tuple defines a unique name for the type definition and creates a secure one-to-one mapping between a type name and a type definition.

A cryptographic hash of the type definition would also define a unique name and a secure one-to-one mapping to the type definition, but we feel that the users benefit from being able to discern event type's owner and the user-friendly name from the name tuple. Also, the type versioning mechanism presented in §4.3 relies on the structure of the event type names.

Notice that the inheritance relationship in Hermes type definitions would also use the complete name tuple including the type owner's identity rather than just the human-readable name.

The Identity of the Type Owner

The type owner's public key defines a globally unique namespace while at the same time specifying the owner of that namespace. Because the namespace is globally unique, accidental name collisions are unlikely assuming that the namespace owner is able to avoid name collisions within its own namespace. A type definition within a namespace is valid only if the signature of the type definition can be successfully verified with the public key from the name tuple. Because the signature can only be created with the type issuer's private key, malicious users are not able to forge the link between the name and the type definition, and therefore are unable to introduce forged type definitions into the system.

Using the type owner's public key to sign the event type definition implies that the key pair must be long-lived in the system, so that the event type can be managed during its lifespan. This must be considered when types are being deployed in the system. Instead of using a key associated with a human principal, the types can be created with a key owned by the domain's access control server, or a threshold subject (See §2.6.4).

Another alternative is to create a temporary key pair just for creating the type definition. That key pair is be used to sign the type definition and delegate all management rights immediately to another principal (We will discuss the delegation of type management duties in §4.3.3). The type management rights would probably be delegated again to the domain's access control service or a threshold subject. This approach is essentially the same as the previous two alternatives with the exception of using a temporary key pair to create the type and then for delegating all access and management rights to the domain or a threshold subject.

Human-Readable Name

Where the type owner's identity defines a globally unique namespace, the human-readable name can be used to build naming hierarchies within that namespace. For example, the reverse-DNS naming scheme used in naming Java packages [Sun99] produces hierarchical names like `uk.gov.pito.Numberplate`. Naming hierarchies enable the event type owner to express a semantic structure among multiple related event types, e.g. all type definitions related to a single publish/subscribe application can be grouped together. Also, including information about the type issuer in the type name will help the application developers to remember who has deployed the type they are working with. We would assume that an integrated development

environment supporting event type definitions would hide the public key part of the name tuple from the user in order to make the interface more user-friendly. Therefore the human-readable name should provide the developer with as much information as possible.

In the congestion control example application described in §3.7 PITO is the owner of the *Numberplate* event type. PITO's identity is signified by P_{PITO} and the full name of the event type is `uk.gov.pito.Numberplate`. Therefore the name tuple for the *Numberplate* event type is [P_{PITO}, `uk.gov.pito.Numberplate`] (we will refer to the event type with *Numberplate* most of the time in order to save space).

4.2.2 Digital Signature

The digital signature on an event type definition is used to verify the authenticity and integrity of the type definition, i.e. by verifying the signature one can ascertain that the type definition has not been tampered with and that it has been created by the signer. The signature is generated over all of the items in the type definition, including the name tuple, i.e. it binds the name tuple to the type definition in a cryptographically secure fashion. This binding allows users, for example, to lookup type definitions from a type registry without having to worry about forged types, because they can verify the authenticity and integrity of the type definition for themselves. The digital signature also allows event brokers to trust event types presented to them by event clients.

4.3 Type Management

We must assume that type definitions in a large-scale publish/subscribe system need to evolve during their lifetime either because of mistakes made in the original definition of the event type or because of changing application requirements during the event type's lifetime. We must also assume that multi-domain publish/subscribe systems must remain operational at all times, because it is very difficult to schedule downtime when multiple domains are running multiple mission critical applications on the same shared infrastructure. Therefore, a type manager must be able to deploy an updated version of an event type definition while the system is still running without disrupting existing clients that are still using the old version of the same type definition.

We expect types to be managed only in the creating domain, but the delegation mechanism places no such constraints. That is, a type owner is able to delegate type management rights to principals in other domains as well. This is important when a domain that created an event type wants to leave the shared publish/subscribe system, but there are still clients using that type. In such a scenario the type owner can delegate all type management rights to some other domain in the system. Another alternative would be for some other domain to recreate a new event type with the same attributes, but this would disrupt the operation of the publish/subscribe system, because the name of the new event type would be different.

Type managers can apply the following type management operations on event type definitions:

i. add attribute,

ii. remove attribute,

iii. rename attribute,

iv. change attribute type.

Unlike the *change attribute type* operation the *rename attribute* operation is not equivalent to a sequence of *remove attribute* and *add attribute* operations, because the rename operation is able to maintain a semantic link between the attribute's old and new name. We elaborate on this in §4.3.2 where we discuss support for renaming attributes in event types while maintaining the semantic link between the two names.

The type management features described above are provided in order to allow a type manager to carefully change an event type definition. We expect that after the release of a new version of a type definition the event clients using that type definition will migrate to the new version in stages: the event clients that are from the same domain as the type manager will probably migrate early on while event clients in other domains will take a longer time before migrating. We also expect that some clients might never adopt the new version. Therefore we feel that it is unrealistic to expect that all event clients are able migrate to a new event type version at the same time.

We describe in §4.3.2 a mechanism for translating published events from one version to another. This mechanism is quite brittle and will result in disconnected event clients if the type manager is not careful when changing the event type definition.

The type management features are provided as a way for the type manager to slowly evolve the event type definition to match new requirements while allowing existing event clients to continue using the current event type version. When changing an event type definition with a number of event clients using the current version of the type, the type manager is responsible for making only changes that allow the event clients to continue to communicate with each other. The type manager should never introduce changes that will prevent a set of event clients from communicating with each other if some of them migrate to the new version of the event type unilaterally.

In cases where there is a risk of event clients losing connectivity, or a slow migration is not possible for some other reason, a completely new event type should be defined instead of trying to update the existing event type which would result in event clients loosing connectivity with each other. In such cases one can deploy type translator clients that will subscribe to the old event type, translate it to an instance of the new event type and publish that instance.

Alternatively type managers can decide to only extend existing types. That is, the type manager can add new attributes or rename existing attributes, but it must never remove an attribute or change its type. By only ever extending the set of attribute, the type manager can guarantee that the events will always contain all attributes expected by a subscriber. On the other hand the type manager cannot force the publishers to include semantically valid values for attributes that they do not care about. That is, a publisher can always set the attribute value to `null` or to some other *undefined* value.

The following two sections describe the changes to the secure event type definitions that we have made in order to support type management in a running publish/subscribe system. The third section will describe how the type owner will be able to delegate management duties to other principals.

4.3.1 Version Number

In order to allow multiple versions of an event type to coexist in the publish/subscribe system simultaneously we propose to include a version number in the name tuple effectively making it a 3-tuple: [*type owner's identity*, *name*, *version number*].

A new version number is created for each new version of a type definition that is deployed. Since the version number is part of the name tuple, changing it effectively renames the event type. This means that introducing an updated event type into the system does not interfere with existing clients, because they have subscribed to and/or publish events of a different type name (i.e. the same type name, but with a different version number).

Existing clients will continue to use the old type until they have been explicitly modified to subscribe to and publish the new version of the updated event type definition. New clients, implemented against the updated type definition, can use the new type immediately.

We assume that clients have to be manually modified to be able to use an updated type definition. The manual modification might be as simple as adding the new type definition to the client's configuration files, or it might require more substantial changes to the client. Notice that under the current assumptions clients subscribing to the new version of the event type will not receive events published as instances of the old version of the same event type. We will address this shortcoming in §4.3.2 by adding *type version translation* to our proposed scheme.

If there is only one type manager, i.e. the type owner, for a given event type, the type manager is free to use a traditional, monotonically increasing version numbering scheme starting from 0. Because there is only one type manager there is not risk of version numbers collisions.

When there are more than one type manager for a given event type, we propose using a version number scheme based on collision resistant values in order to avoid having to serialise access to a version number counter. If we were to use a counter-based version numbering scheme with more than one type manager, each type manager would have to ask an authoritative party for the next version number in order to avoid version number collisions. This would be

both slower and more expensive, not to mentioned more difficult to implement, compared to collision resistant version numbers in a distributed setting. Note that even with multiple type managers the initial version of an event type can always have a version number 0, which will also identify the original version of an event type.

In MAIA we have implemented event type version numbers as *Universally Unique Identifiers* (UUIDs) [Ope97]. A UUID is a 128-bit value that is guaranteed to be unique from all other UUIDs until the year 3400. Instead of UUIDs we could also use some other collisions resistant value, for example, the cryptographic hash of the type definition. Notice that we are only concerned with accidental collisions instead of malicious collisions in version numbers. Therefore we could use MD5 [Riv92] or SHA-1 [FIP02] as a collision resistant hash function even though these two algorithms are no longer considered to be secure against malicious attacks [WY05, WYY05].

One of the advantages of UUIDs is that they are generally faster to create than cryptographic hash values. On the other hand a cryptographic hash value can be forced to collide with an existing value. That is, by calculating the hash value from the attributes set the version number of an event type definition will be the same as the version number of another version of the same type definition with an equal attributes set. In contrast, a UUID-based version number scheme will always produce a unique version number regardless of the attribute set. In a hash based scheme, if a type manager releases a new version of an event type definition that is equivalent to a previous version, the two versions would have the same version number and therefore the same name. Notice in order to generate colliding hash values, the version number must be generated by hashing only the attribute set. The calculation must not include any other fields from the type definition, e.g. a creation timestamp or the *credentials* field that we will introduce in §4.3.3, because these fields would bind the version number to a specific principal who updated the type definition or to a point in time when the update happened thereby preventing other principals from generating colliding version numbers.

We felt that equivalent version numbers for equivalent event type definitions would not provide us with any further advantage compared to type version translation described in §4.3.2. Therefore MAIA implements UUID-based version numbers.

It was also suggested to use the digital signature on a type definition as the version number. Unfortunately this results in a chicken and egg problem, because the name tuple that includes the version number is included in the type definition that is being signed. That is, we cannot create the name tuple with a version number, because the version number is based on the signature and we cannot create the signature, because the signed data must include the name tuple. Remember from §4.2.1 that including the name tuple in the type definition creates a link between the name tuple and the type definition, which allows us to verify the authenticity and integrity of both the type definition and the name tuple.

Compared to a counter-based version number, collision resistant schemes lack a clear time line. One cannot order collision resistant version numbers, because they are basically uniformly

random numbers and one can not say if one version precedes another one or not. With a counter-based scheme it is obvious that version n precedes $n + 1$. In order to provide a time line for type definition versions we can add the preceding version number to the type definition. With the help of the reference to the preceding version we can create a partially ordered version tree. We feel that there is no need for partial ordering of event type versions and have therefore not added a reference to the preceding version to our secure event type definitions.

With a version number the name tuple for the *Numberplate* event type is a 3-tuple: [P_{PITO}, `uk.gov.pito.Numberplate`, 1234]. Notice that the real version number would be a 16-byte UUID number that in its canonical form would be a 36-character hex string, e.g. 550e8400-e29b-41d4-a716-446655440000. We will be using four character integers to represent version numbers in the rest of this dissertation in order to make the text more readable.

4.3.2 Type Version Translation

A side effect of adding the version number to the name tuple is that subscribers will not receive events from publishers that are publishing instances of a different version of the same event type. This presents a real problem when either the publisher or the subscriber updates its event type version unilaterally. Event clients might migrate to the new version of the event type in cases where the event clients are in different domains, e.g. the event clients in the type manager's domain might be given access to the new version earlier than event client in other domains. Even if all event clients were migrating to the new version at the same time it would be improbable that they would do so at exactly the same moment in time. If a publisher and a subscriber are using different versions of the same event type the publisher and subscriber will lose connectivity with each other, because the different event type versions are treated as unrelated event types.

Instead of routing events of specific event type versions, we translate the published event to a *transit time event* at the PHB after the event has been type-checked by the PHB against the version of the type definition used by the publisher. The transit time event is simply a collection of name-value pairs that were copied from the publication. It also includes the type owner's identity and the human-readable name from the name tuple, but not the version number.

The transit time event is then routed through the broker network to all subscribers of that event type, regardless of the version of the type definition used in the subscriptions. The SHB then translates the transit time event to an instance of the subscriber's version of the type definition. Attributes that are present in the transit time event, but not in the subscriber's version of the type definition are ignored. Similarly attributes that are not present in the transit time event, but are defined in the subscriber's version of the event type are set to `null` (or *undefined*) in the event instance, as shown in Figure 4.2.

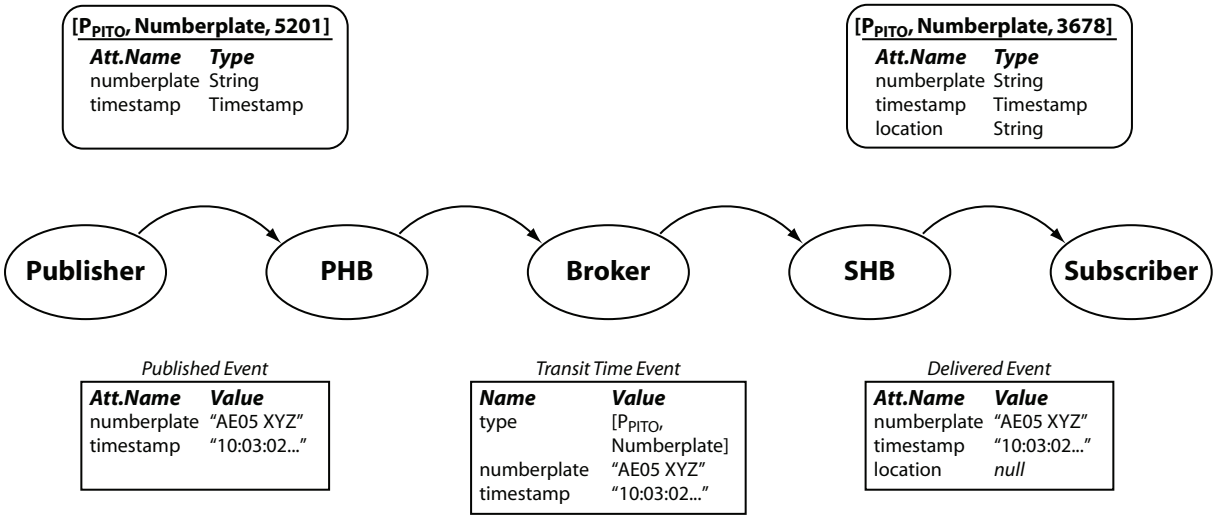

Figure 4.2: Translation to and from transit time events.

Renaming Attributes

The operation for renaming attributes in a type definition results in problems when translating an instance of one version of an event type to an instance of another version. More specifically, the publisher might use a version of the *Numberplate* type definition that refers to an attribute with the name `license-plate` while the subscriber is using another version of the same event type that refers to semantically the same attribute with the name `numberplate`. The transit time event created from the publication will also refer to the attribute with the name `license-plate`. The SHB translates the transit time event to an instance of the subscriber's version of the *Numberplate* event type, but because that version of the event type uses the name `numberplate` instead of `license-plate`, the SHB sets the value of `numberplate` attribute to `null` before delivering the event to the subscriber.

In order to allow the renaming of attributes in event type definitions without losing the semantic link in the process, we add a unique identifier to each attribute definition. We replace the attribute 2-tuples in event type definitions consisting of a name and a type with a 3-tuple consisting of a name, a unique identifier, and a type, e.g. [`numberplate`, 1234, String]. When renaming an attribute, the name in the 3-tuple changes, but the unique identifier and the type stay the same.

The attribute definition defines a mapping between the attribute name and the unique identifier. The name has to be unique within the context of that particular version of the type definition whereas the unique identifier has to be unique within the context of all versions of the type definition. A unique identifier therefore identifies a single attribute among all the attributes defined for that event type in all its different versions.

Similarly to event type version numbers described in §4.3.1, the attribute identifiers in the first version of an event type definition can be created as sequence numbers, i.e. 0, 1, 2 etc.,

because there is not risk of collisions. Assuming that the type definitions change very little during their lifetimes, we can guarantee that we create the least amount of overhead by assigning the original attributes the shortest possible identifiers. The attribute identifiers in following versions of the event type can be either sequence numbers or collision resistant values depending on whether there is more than one type manager creating new versions of the event type simultaneously, as was the case with event type version numbers.

Instead of using a UUID we could use some other identifier generation scheme that would be collisions resistant in the relatively small identifier space of an event type definition. The attribute identifier has to be unique among all the attribute identifiers of all the versions of that single event type definition. Assuming ten different versions of an event type where each version has a completely new set of ten attributes we would require one hundred identifiers. Using a 16-byte UUID for each attribute seems wasteful considering the number of unique identifiers that need to be created (it would be safe to assume that most event types will have only a few versions with mostly the same set of attributes in each version). One possible solution would be to fold the 16-byte UUID into n bytes by XORring the n-byte segments into one single value. Another approach would be to take the n-byte remainder of the generated UUID and use that as the identifier.

The length of the identifier can be varied based on the expected number of attributes for a given event type definition. In both approaches it is important to verify that the resulting values are uniformly random and collisions resistant. The length of the identifiers can also vary from version to version which allows the type owner to switch from scheme to scheme depending on the current circumstances.

The publish/subscribe clients refer to attributes using attribute names, whereas the intermediate brokers use the unique identifiers. Before routing the event through the broker network the PHB creates a transit time event from the publication by translating the attribute names to UIDs based on the publisher's version of the type definition. The SHB translates the UIDs back to attribute names based on the subscriber's version of the type definition before delivering the event to the subscriber.

The transit time event uses UIDs instead of names to refer to attributes. The unique identifiers guarantee that attributes are always unambiguously identifiable regardless of their names in any given version of the event type. This allows type managers to rename existing attributes and reintroduce old attribute names while maintaining interoperability between different versions of a type definition. For example, in Figure 4.3 the publisher uses the version 4799 of the type *Numberplate* with the name `license-plate` for the attribute with UID 9525, while the subscriber uses the version 8516 of the same type with name `numberplate` for the same attribute. Although the names for the attribute are different for the publisher and the subscriber, the UIDs are the same, which allows the subscriber's local broker to deliver the attribute to the subscriber as `numberplate`. Similarly a new version of the type definition derived from version 4799 might include a new attribute with the name `license-plate`, but with a UID

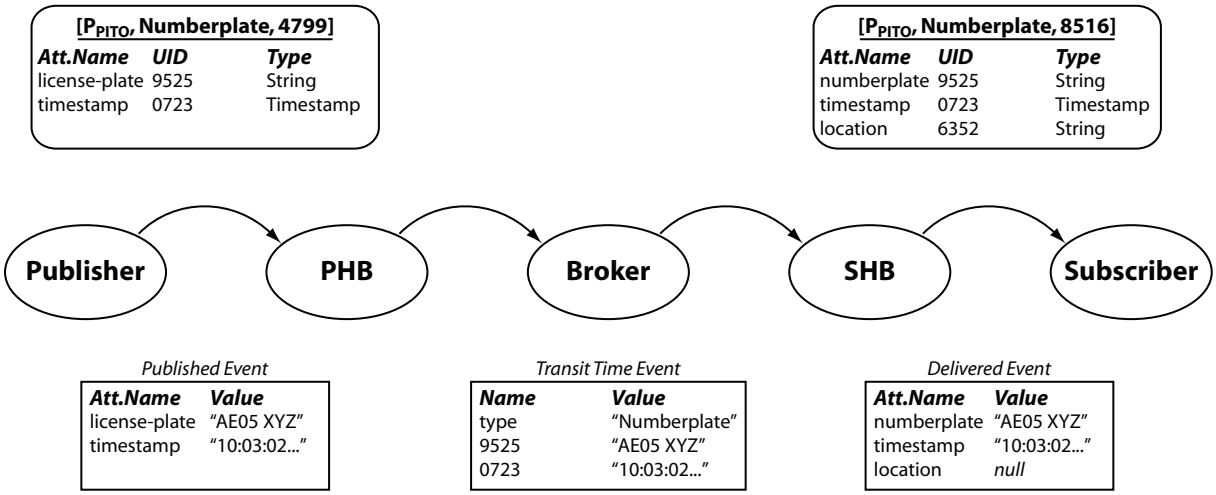

Figure 4.3: Translation to and from transit time events with attribute UIDs.

3879. Because the UID (3879) differs from the UID (9525) of the original `license-plate` attribute, there is no risk of confusing one attribute with the other when converting instances of the type to and from transit time events.

Table 4.3 shows the *Numberplate* event type definition with a name tuple including a version number and attribute tuples that include a unique identifier.

4.3.3 Authorisation Certificates

In a scalable system, the management of an event type cannot be the responsibility of a single principal, i.e. the type owner. We expect that in an Internet-scale publish/subscribe system event types are long-lived and that type owners can leave the system. For example, a domain that owns an event type, that is used in the multi-domain publish/subscribe system by other domains, leaves the system, but wants to allow the other domains to carry on using that event type. Therefore the system must allow delegation of type management duties to one or more type managers. Unfortunately the secure event type definitions rely on the fact that the identity in the name tuple verifies the digital signature on the type definition. Thus, only the principal identified by the public key in the name tuple, i.e. the type owner, is able to create a digital signature that is verifiable with that public key. Because of this only the type owner is able to deploy updated versions of the type definition. If another type manager were to edit the event type, sign it, and reintroduce it to the system this link would be broken, because the type manager is unable to sign the type definition with the type owner's private key.

To overcome this limitation we propose using signed capabilities to delegate type management duties to other principals. The type owner issues a capability to each type manager authorising the type manager to issue new versions of the specified event type. The capability includes both the issuer's (i.e. the type owner's) and the subject's (i.e. the type manager's) identities, thereby creating a link between the two principals.

Name	Value
name	[P_{PITO}, `uk.gov.pito.Numberplate`, 8516]
attributes	[`numberplate`, 3265, String]
	[`timestamp`, 9058, Timestamp]
	[`location`, 3467, String]
signature	The type owner's digital signature containing all of the above elements

Table 4.3: The *Numberplate* event type definition with a version number and attributes with unique identifiers.

Field	Description
name	A tuple consisting of the type owner's identity, a human-readable name and a version number
attributes	A set of attribute definitions
credentials	A set of signed capabilities
signature	The type owner's digital signature containing all of the above elements

Table 4.4: A secure event type definition with a *credentials* field.

The type manager includes its capability chain in the type definition and signs the definition with its private key. A verifier can now link the signature to the name tuple by following the capabilities chain from the type owner's identity in the name tuple to the type manager's identity in the capability, and verify the signature with the type manager's public key. The capability chain links the type manager to the type owner and allows the verifier to trust the type manager's signature. We add a *credentials* field to the secure type definition to hold the capabilities, as seen in Table 4.4.

The capability can also authorise the subject of the capability to delegate a subset of its authority to a third party. Therefore the certificate chain may consist of more than one certificate.

Figure 4.4 depicts three different cases of type management. In the first column the type owner, P_1, has created or updated the type definition and signed it. The type owner's signature is directly verifiable with its identity present in the name tuple. Therefore there is no need for a capability and thus the credentials field in the type definition is left empty. In the second column the type manager, P_2, has updated the type definition. P_2 includes in the updated type definition a capability that has been issued by P_1 and authorises P_2 to manage the type definition. The capability links the signature, $S(P_2)$, to the type owner's identity P_1. The third column is similar to the second column, except in this case the capabilities chain linking $S(P_n)$ to P_1 consists of more than one capability.

The set of credentials is there to link the current signature to the identity in the name tuple. The type manager updating the type definition always replaces the previous set of credentials

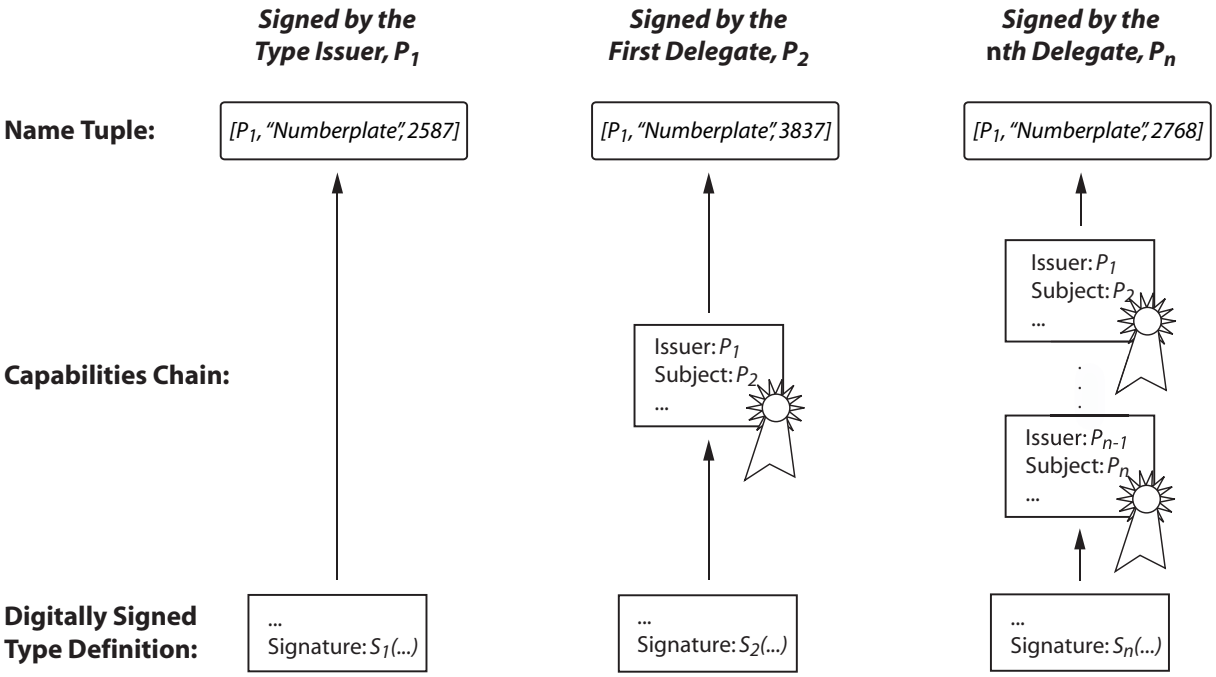

Figure 4.4: Verifying the name-signature link with and without a capabilities chain.

with a set of credentials that link it to the name tuple. In the default case where the type owner issues the type definition and manages all updates, there is no need to include any authorisation certificates (See the first column in Figure 4.4).

The authorisation certificates are delivered to the type managers out-of-band, i.e. independent of the type definitions. Out-of-band delivery enables the type owner to grant authorisation to a principal even after the type definition has already been deployed. If the capabilities were embedded in the type definition, the type owner would have to update the type definition every time she wanted to change the access control policy to issue or revoke type management rights. This would result in otherwise unnecessary versions of the type definition, and maintaining an up-to-date set of authorisation certificates as part of the type definition would be close to impossible if the type managers were authorised to further delegate type management rights to other principals.

We will discuss access control based on capabilities in more detail in Chapter 5, where we will present an access control architecture for multi-domain publish/subscribe systems based on signed capabilities and delegation.

4.4 Modifications Made to Hermes

We made a number of changes to the original Hermes design in order to add support for secure event types. Some of the changes were necessary for Hermes to be able to support secure event types. Other changes were made possible by secure event types and allowed us to simplify the

Hermes design. The following sections describe in detail the more important changes that were done.

4.4.1 Type Storage

The original Hermes design uses a *distributed hashtable* (DHT) [RD01a] to store event types in the broker network. The name of the event type is used as a key when inserting a type definition into the DHT. The unreliable nature of structured overlay networks demands that the stored type definition be replicated among multiple nodes. That is, because the nodes of the overlay network may leave at any time, not to mention the possibility of node and network failures, the content stored at a specific node must be replicated to other nodes in order to guarantee the availability of the content with high probability. Maintaining the DHT thus results in a lot of unnecessary network traffic when content is copied to replica nodes during inserts, and when nodes join and leave the overlay network. Even with replication, the DHT can only provide availability with high probability based on the number of replica nodes. In the worst case the requesting node is left stranded from all replica nodes after a network partition and thus not able to access the content.

In addition to performance issues Hermes does not provide any integrity or authenticity guarantees for type definitions. The integrity of type definitions in a multi-domain publish/subscribe setting must be protected so that accidental and malicious modifications do not go unnoticed. Otherwise implementing an access control based on those type definitions is doomed to failure. In Hermes all participants are expected to trust the event service to protect the integrity of event type definitions. In a multi-domain environment it is unlikely that all participating domains are willing to trust the other domains with their event type definitions.

The self-certifiability of secure event types enables both event clients and event brokers to verify the authenticity and integrity of type definitions. Because the authenticity and integrity of type definitions can be verified it is no longer necessary for all participants to trust the event service to protect the integrity of a type definition. Type definitions can be stored in the event service as is the case with Hermes with the knowledge that any tampering by any of the event brokers that implement the type storage will be noticed.

We argue that developers implementing publish/subscribe applications that handle specific event types need to have the definitions of those types available to them during development time. The type definitions would be delivered to the developers out-of-band, e.g. as downloads from the type owner's web page or a type repository. Since the type definitions are part of the development process it would be simple to include them in the packaging of the publish/subscribe applications. The client would then be able to pass the type definition on to the local broker as a part of an appropriate publish/subscribe request (`advertise` or `subscribe`). The broker would then verify the authenticity and integrity of the client-provided type before executing the client's request by verifying the digital signature and possible authorisation certificates.

Function	Attributes
advertise	*(*`type_name`*)*
unadvertise	*(*`type_name`*)*
publish	*(*`event_instance`*)*
subscribe	*(*`type_name`, `filter`, `callback`*)*
unsubscribe	*(*`type_name`, `callback`*)*
addEventType	*(*`type_def`*)*
modifyEventType	*(*`type_def`*)*
removeEventType	*(*`type_name`*)*

Table 4.5: The Hermes event client API.

Only the local brokers need to do type-checking. The PHB type-checks the submitted publication before it is routed through the broker network. Similarly the SHB type-checks the subscription filter before passing the subscription on to the broker network. The SHB also type-checks all publications against the subscriber's version of the type definition. With publications the SHB can type-check a given publication once against each version of the type definition and use that result for all subscribers that have subscribed with the same type definition version. The intermediate brokers assume that the publications and subscriptions have already been type-checked by the local brokers, thus only the local brokers need to be aware of type definitions.

By relying on publish/subscribe clients to provide type definitions to local brokers we remove the need for maintaining a type repository in a DHT, thus lowering the amount of system internal network traffic and making the broker design more elegant and simpler in general.

4.4.2 API Changes

The new approach to type storage presented in the previous section and the introduction of the name tuple result in changes to the Hermes API, shown in Table 4.5 (the Hermes API is described in more detail in [Pie04].

Because type definitions are not installed in the publish/subscribe system any more, as explained above, there is no need for a specific type management API with the operations `addEventType`, `removeEventType`, and `modifyEventType`. Type definitions are provided to the brokers by the clients as parameters to the `advertise` and `subscribe` operations. In both cases the old API operations referred to the event type with the type name. These operations are provided in the new API as well in order to allow for cases where the client hosting broker has cached the event type definition and therefore the client is able to refer to it by name. Table 4.6 defines the new API implemented in MAIA.

Note that we have removed type management operations from the API exported by the event brokers, but this does not mean that one could not implement a type registry in the broker network as has been done in Hermes. We merely feel that the type registry functionality should

Function	Attributes
advertise	*(*`type_def`*)*
advertise	*(*`type_name`*)*
unadvertise	*(*`type_name`*)*
publish	*(*`event_instance`*)*
subscribe	*(*`type_def`*,* `filter`*,* `callback`*)*
subscribe	*(*`type_name`*,* `filter`*,* `callback`*)*
unsubscribe	*(*`type_name`*,* `callback`*)*

Table 4.6: The MAIA event client API.

be independent from the publish/subscribe functionality.

4.4.3 Message Routing

In addition to using the event type name as a key in the DHT when storing type definitions, Hermes uses the type name as a node id when routing events through the broker network. Hermes chooses a *rendezvous node* from all the broker nodes by hashing the type name in order to create a node id. An event dissemination tree is then created in the broker network by routing advertisement and subscription messages towards the rendezvous node. Because the rendezvous node id is created by hashing the name of the event type, both the PHB and the SHB are able to compute the node id of the rendezvous node from the type's name tuple without any external assistance. Publications are then routed based on the event dissemination tree from the publisher to all subscribers. The Hermes routing algorithm is explained in more detail in [PB02] and [PB03].

Simply hashing the name tuple would result in each event type version having a different rendezvous node, because $h(P||n||v_1) \neq h(P||n||v_2)$. This would result in independent event dissemination trees for each version of an event type and, because of this, in unnecessary routing state and sub-optimal routing performance. Moreover, the event dissemination tree for one version would not be able to reach subscribers of another version of the same event type, e.g. v_1 publications would not reach v_2 subscriptions. Instead of hashing the whole name tuple, we ignore the version number and hash only the public key and the name: $h(P||n)$. This results in a common rendezvous node and optimal routing performance for all versions of an event type allowing, for example, v_2 subscribers to receive both v_1 ans v_2 publications (See §4.3.2 regarding type version translation.).

4.5 Performance

The most significant performance penalty in verifying secure event types is caused by digital signature verification. The other related operations, e.g. SPKI 5-tuple reduction, are very cheap

in comparison. This is clear from the performance results in Table 4.7 that show the time spent for both operations in microseconds. The 5-tuple reductions are over 50 times faster than digital signature verifications. The reduced SPKI certificate chain was five certificates long and the digital signatures were generated with a 1024-bit RSA key. The test runs were executed on an Intel P4 3.2GHz workstation with 1GB of main memory.

Operation	μs	σ
5-tuple reduction	5.533	0.18513
RSA signature verification	291.926	2.60839

Table 4.7: The time in microseconds spent on 5-tuple reductions on RSA signature verifications.

Event types need to be verified when a publish/subscribe client provides an event type to the local broker as a part of an advertisement or a subscription request. Publications refer to an already verified event type and thus do not need to be verified individually.

In a naïve implementation a broker verifies every client-provided event type for every advertisement and subscription request separately. An optimised implementation would cache the verification result of each event type and simply compare the already verified type to the event types in subsequent requests thus avoiding the expensive signature verification. When the type cache is enabled in MAIA the plain and signed type definitions perform equally well as would be expected. This is shown in Table 4.8.

Event Types	μs	σ
Plain types	6.77053	0.47450
Signed types	7.10021	0.83788

Table 4.8: The time in microseconds spent on processing a subscription request at the local broker for plain types and signed types when the type cache is enabled.

The broker can also store client-provided event types locally after verification. This enables the broker to load and verify those event types as part of the broker start-up sequence. As we can assume that the set of event types in use in a publish/subscribe system is relatively static, i.e. the publish/subscribe clients advertise and subscribe to the same event types most of the time, the bulk of the cost of verifying those types is paid in advance while the broker is starting up. The routing performance of a broker is only affected by new types and type versions introduced to the system that have not been verified yet.

The cost of verifying an event type depends on the length of the certificate chain in that event type. Therefore the impact of event type verification can be reduced even more by the broker caching also verified authorisation certificates. The cached certificates can then be used in verifying certificate chains in other event types and event type versions where the certificate

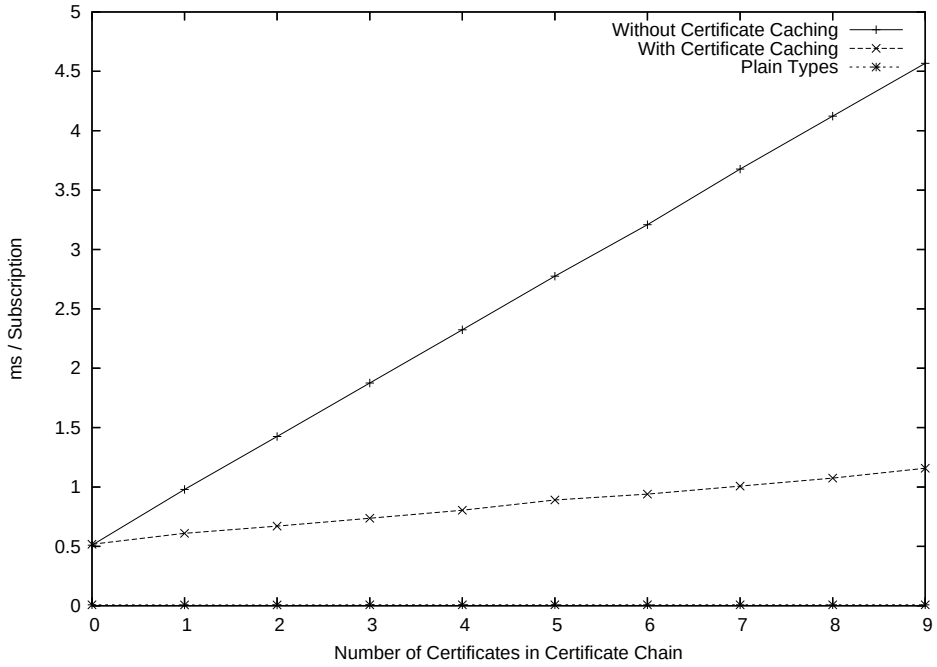

Figure 4.5: Subscription performance with and without certificate caching.

chains contain cached certificates. For example, if a type manager introduces a new version of an already cached event type to the system, the broker can use the cached certificates in verifying the certificate chain from the type owner to the type manager. Figure 4.5 shows how the processing time of a single subscription increases when the length of the certificate chain increases. Notice that all the subscriptions use the same event type and therefore we have disabled the type cache, which would otherwise affect the results. With certificate caching disabled the local broker must verify each certificate and the digital signature on the type definition for each subscription. When certificate caching is enabled the broker verifies the certificates in the chain only once. For the remaining subscription the broker has to verify only the digital signature on the event type definition. The Plain Types graph is included to highlight the price of verifying digital signatures. A subscription for a plain event type takes on average 7.422 μs with a standard deviation of 1.78347 μs. The high standard deviation is the result of a small test run, which is completed very quickly when using plain types. A larger test run was not possible, because the same test run for signed types would have taken too long.

In any real world deployment the type cache would be enabled and we would get performance results comparable to the results in Table 4.8. Nevertheless the timing results in Figure 4.5 show that a certificate cache does provide a performance increase when verifying signed event types that share certificates in their certificate chain.

Although signed event types do affect the time required to process subscription and advertisement requests, we assume that subscriptions and advertisements will represent only a small portion of all the requests made to local brokers and therefore the performance impact should

be relatively small. Even in cases where subscribers and publishers disconnect from the event service quite frequently it is safe to assume that they will usually subscribe or advertise the same event type that they used the previous time they were connected to the event service. In such cases the event type cache will provide comparable performance. Therefore we can conclude that the biggest performance impact is suffered when new event types are introduced to the event service frequently. We expect this to be relatively rare.

Note that adopting secure event types in a publish/subscribe system does not affect the routing performance of the system at all. All type-related operations, e.g. validating the event types and type-checking publications, are performed only at the client hosting brokers either at subscription or advertisement time, or at publication time, respectively.

4.6 Secure Names in Topic-Based Publish/Subscribe

While the bulk of this chapter has been specifically about type-based publish/subscribe systems, the concept of secure namespaces and secure names are useful in topic-based publish/subscribe as well. By prefixing the topic owner's identity to the topic name we get a 2-tuple that is globally unique with high probability (See §4.2.1).

With type-based publish/subscribe the identity in the name tuple was bound to the type definition through the digital signature covering the whole definition including the name tuple. The signature creates a two-directional link that on the one hand guarantees that the type definition is authentic and correct, but it also verifies that the type name is authentic, i.e. that it has been created by the owner of the identity. In the case of topic-based publish/subscribe there is no type definition and no signature that could be used to verify the authenticity of the topic-name. Therefore topic-names with the topic owner's identity prefixed to the name are not useful as such, because their authenticity cannot be verified.

Fortunately, the access control architecture that we introduce in Chapter 5 binds the topic name to the authorisation certificates that grant principals access to the topic, and the authenticity and integrity of those certificates can be verified in a similar manner as we verified the authorisation certificates used in this chapter to delegate type management rights. Thus, we can use secure topic names to implement access control for topic-based publish/subscribe.

4.7 Related Work

Wang et al. present in [WCEW02] a number of security issues in large-scale publish/subscribe systems that need to be addressed before publish/subscribe systems can be deployed in the Internet. The paper covers problems related to authentication, data integrity and confidentiality, accountability, and service availability. We feel that secure names and event type definitions provide a foundation on which to build solutions to the problems they discuss.

Linked local namespaces were first introduced by Rivest and Lamport as part of the SDSI 1.0 specification [RL96]. In SDSI a principal maps local names to the public keys of other principals. For example, the local name *foo* in P_1's namespace is bound to principal P_3, but to P_4 in P_2's namespace. Local names can also be chained, e.g. [P_1, *foo*, *bar*], which points to the principal known as *bar* by the principal who is known as *foo* by P_1.

Our proposed naming scheme for secure event types borrows from SDSI in creating globally unique namespaces by using public keys as the root of the namespace. The appearance of secure names resembles SDSI's linked local names, but in fact they are different. In SDSI all names always point to a principal with its own key pair. In our case one could think of the human readable part of the secure name as a local name in the type owner's namespace, that points to the event type definition. But this would be incorrect, because the event type is not a principal. Similarly one could think of the version numbers as local names in the event type's namespace, but again this would not be accurate, because versions are not principals either. In SDSI one can reduce a chain of names by replacing the public key and a local name with the public key that represents the local name. With secure names the analogy would be to replace the type owner's public key and the human readable name with the event type definition's public key, which is not allowed in the current scheme. So, while secure names have been inspired by and appear similar to SDSI's linked local names, in actual fact they are different.

The type management approach with multiple concurrently active type versions in the system was inspired by schema evolution in object-oriented databases. For example, the ORION object-oriented database [BKKK87] also uses unique version numbers instead of names to identify schema entities in a similar way that we use UUIDs to identify attributes in event types. The indirection created by the use of version numbers allows the name of the entity to change from one version of the schema to another while maintaining a semantic link between the two entities with different names in the two versions of the schema.

4.8 Summary

This chapter presents a model for secure event type definitions in type-based publish/subscribe systems. The scheme provides self-certifiable type definitions that allow both event clients and brokers to verify the authenticity and integrity of a type definition. Although our design is based on Hermes, it is applicable to type-based publish/subscribe systems in general and to topic-based publish/subscribe systems with certain limitations as described in §4.6. We presented the work originally in [PB05].

We feel that secure names and event type definitions provide the foundation for a secure publish/subscribe middleware. Other services like access control can then be built on this foundation. For example, in the case of access control we can bind access rights to topic, type and attribute names, because we can trust those names to be unforgeable and unique. At the same

time, digitally signed type definitions allow us to place policy information in the form of SPKI authorisation certificates inside event definitions.

In addition to secure event types, we have also introduced a scheme for managing event types in a large-scale publish/subscribe system. We feel that large-scale publish/subscribe systems must be able to run continuously regardless of type management operations. Our approach enables type managers to update existing type definitions transparently without affecting existing clients. We also support the delegation of type management duties to other principals, which we see as an equally important feature when considering the expected lifetime of event types in a large-scale, highly decentralised publish/subscribe system.

CHAPTER 5

Access Control

In Chapter 4, we introduced a scheme for creating unique and verifiable names for event types and attributes. In this chapter we will leverage that contribution in designing a decentralised access control system for multi-domain publish/subscribe systems. We use signed capabilities to describe the global access control policy of the multi-domain environment in a decentralised fashion. That policy refers to publish/subscribe network names, event types and attributes, which must be both unique and unforgeable for the access control policy to be unambiguous. We use a similar naming scheme for network names as we did in Chapter 4 for event types, i.e. we prefix the coordinating principal's identity to the human-readable network name in order to guarantee its uniqueness and verifiability. For example, the PITO coordinated UK Police Network is named [P_{PITO}, `UK Police Network`].

In addition to secure names we also utilise the signed event type definitions introduced in Chapter 4 to store credentials related to event type deployment and management operations. For example, the credentials authorising a type owner to deploy a new event type on the publish/subscribe system are included in the type definition being deployed.

Incorporating the credentials in the type definition allows all nodes in the publish/subscribe system to verify that the type owner has been authorised by the coordinating principal to deploy new event types on the publish/subscribe system. We use the same mechanism when authorising the deployment of new event types and the management of existing types.

The tight coupling of type management credentials and type definitions prevents us from updating the credentials after the type has been deployed. But this is only a problem when the credentials embedded in the type definition expire. In such a case the type definition can be seen as expired and a new version of the type definition must be deployed with fresh credentials. We will discuss credential expiration and revocation in detail in Chapter 6.

In a decentralised, multi-domain publish/subscribe system we need to control access to two types of resources: the shared infrastructure (i.e. the event service / broker network) and the event types that have been deployed on that infrastructure. The following two operations have to be authorised by the owner of the infrastructure:

i. event brokers and event clients, joining and accessing the broker network

ii. type owners deploying new event types on the broker network

And the following two operations have to be authorised by the owner of the event type:

i. event clients and brokers accessing event types and attributes in those types through the publish/subscribe API either by publishing or subscribing to an event type

ii. type owners delegating type management duties to other principals

Connecting to the broker network and accessing event types are both actions that are executed by clients when they connect to the broker network to publish and to subscribe to events. The other two actions, deploying event types and delegating type management duties, are executed only when new types or new version of existing types are to be deployed on the infrastructure.

We propose a common approach to access control in publish/subscribe systems for all five actions where access control decisions are ultimately rooted at the resource owner. Employing signed capabilities and distributing the access control policy management, decision making, and credential management over all resource owners enables the access control architecture to scale up in a decentralised environment consisting of multiple independent administrative domains. The architecture also allows domains to implement an access control policy management approach of their choice, for example, role-based access control or access control lists.

The rest of this chapter is organised as follows. We introduce our access control model in §5.1. This section describes how access rights are delegated to domains, and domain members. In §5.2 we discuss the various resources that are protected by the access control system, what kind of access operations those resources support, and how access rights to those resources are represented in the access control system. We describe how a principal's authority is verified in the system in §5.3. An important part of decentralised access control is the concept of root authority, i.e. which principal is ultimately responsible for all access control decisions for a given resource. We discuss in §5.4 how the resource owner is able to delegate root authority, i.e. all rights to a resource, to another principal and how this feature can be used to manage event types etc. within a domain. While this dissertation focuses on type-based publish/subscribe, we give a short overview on how our access control system could be applied to a topic-based publish/subscribe system in §5.5. There exist other proposals for access control in publish/subscribe systems, which we present in §5.6. Finally, §5.7 gives a short summary of the whole chapter.

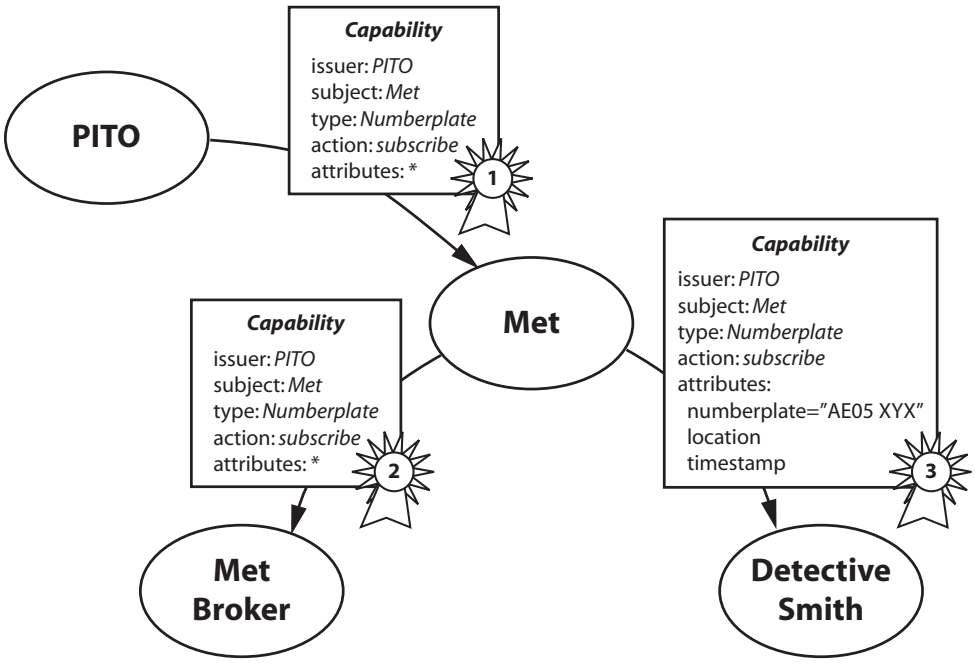

Figure 5.1: Capability 1 authorises the Met domain to subscribe to all attributes of the *Numberplate* event. Capabilities 2 and 3 delegate a subset of this capability to both the Met Broker and Detective Smith.

5.1 Access Control Model

We propose a two-tiered access control model for multi-domain publish/subscribe systems. On the first tier a resource owner authorises a domain to access a given resource. On the second tier the domain delegates its members a subset of that authority. For example, in Figure 5.1 following our example scenario, the type owner, PITO, has authorised the Metropolitan Police domain to subscribe to *Numberplate* events. The Metropolitan Police domain has delegated a subset of that authority to the Met Broker and Detective Smith. The Met Broker shares the authority granted to the Met domain, while Detective Smith's authority is limited to a specific set of *Numberplate* event attributes with a mandatory filter on the `numberplate` attribute. One can think of event brokers as proxies of their domain, representing the domain in the publish/subscribe system. Therefore, the brokers share the domain's authority. We will discuss authorising event brokers in more detail in §5.1.3.

5.1.1 Authorising Domains

To authorise a domain to access a resource the resource owner issues a capability to the domain. In practise the *access control service* (ACS) of each domain represents that domain and therefore the capability is issued to the ACS.

The capability specifies the authority, A_d, that has been granted to the domain. In addition to the access rights, the capability also authorises the domain to further delegate a subset of

the granted authority to members of the domain. The authority granted to the domain by the resource owner is always a subset of the resource owner's authority: $A_d \subseteq A_r$.

The ACS issues capabilities to the members of the domain in order to authorise them to access a resource. When issuing capabilities to domain members the ACS implements a domain-internal access control policy that is completely independent of the resource owner, i.e. the resource owner cannot affect the access control policy. This is typical of discretionary access control systems.

The authority granted to a domain member, A_m, is always a subset of the authority granted to the ACS, A_d, and therefore a subset of the resource owner's authority, A_r, i.e. $A_m \subseteq A_d \subseteq A_r$. Notice that the subset relation is enforced by the 5-tuple reduction rule, as described in §2.6, that collapses a chain of capabilities by taking the intersection of all the authority fields. Therefore the authority field in a capability lower down in the chain of capabilities can be a superset of the authority of some capability higher up in the chain, while still maintaining the subset property described above. We will rely on this property when we authorise event brokers in a domain in §5.1.3.

Domain members are event clients and brokers, and sub-domains, as stated in §3.2. Event clients access the publish/subscribe system through a trusted local broker in order to implement a distributed application with other event clients. In contrast both event brokers and sub-domains implement infrastructure and are there to facilitate the event clients. The following three sections will describe in detail how a domain can delegate authority to each of the three types of domain members.

5.1.2 Authorising Clients

The domain delegates authority to an event client by issuing a capability to the client, which specifies the delegated access rights. As stated in the previous section, the delegated authority is a subset of the domain's authority. We do not expect event clients ever to be authorised to further delegate their authority, because delegation and access control policy management are the responsibility of the domain (i.e. the ACS).

The domain-internal access control policy specifies what access rights should be delegated to each client. The access rights of two event clients can vary significantly, e.g. two detectives in the Metropolitan Police domain are likely to be working on different cases which require access to different information and therefore different events. Enforcing such an access control policy centrally, i.e. by the resource owner directly, would not be scalable. Also, in most cases the resource owner simply cannot understand the domain's internal policies and therefore is not able to specify a global access control policy.

We expect the ACS to issue very fine-grained certificates to event clients that authorise the client to access a single resource in a very specific manner. This allows the system to implement the *principle of least privilege* [SS75], according to which principals should be granted only

those privileges that are required for the task at hand. We also assume that the access control policy for event clients is very dynamic and relies on environmental predicates and certificates issued by other parties. For example, in Figure 5.1 Detective Smith has been authorised to subscribe to events related to a single numberplate. We should hope that Detective Smith is granted this authority only after showing the ACS a court order authorising her to monitor the movements of a specific numberplate in London.

5.1.3 Authorising Event Brokers

The event brokers in a domain form the domain-internal event service and connect that event service to the shared, multi-domain event service. The purpose of the event brokers is to export the publish/subscribe API to the domain's event clients allowing the clients to publish and to subscribe to events over the multi-domain publish/subscribe network. Therefore, we can assume that an ACS would grant all the event brokers in the domain the same authority that it has, i.e. $\forall \beta \in B : A_\beta = A_d$, where B is the set of event brokers in the domain. If the brokers share the domain's credentials, then an event client is free to connect to any event broker in that domain without having to worry that the event broker is not able to access a given resource (i.e. publish/subscribe network, or event type) for lack of access rights. We can guarantee that $\forall \beta \in B : A_c \subseteq A_\beta$ when $\forall \beta \in B : A_\beta = A_d \wedge A_c \subseteq A_d$.

We assume in this chapter that all the brokers forming the event service, i.e. the broker network, are trustworthy and that we need to enforce access control policy only at the edges of the broker network. Addressing access control only at the edges of the broker network results in a system where any domain can circumvent the access control policy simply by not enforcing the policy at the domain's event brokers. More specifically, any broker that an event is routed through is able to read and alter that event. We will address this problem in Chapter 7 by using encryption to enforce access control inside the broker network. We will therefore require already in this chapter that event brokers have the same access rights as the event clients they are hosting in order to be able to fulfil the clients' requests, as described above. For example, a broker will not be able to fulfil a client's subscription request for *Numberplate* events if the broker itself is not authorised to subscribe to *Numberplate* events and access the same set of event attributes.

There are two approaches that a domain can implement in order to delegate access rights to all its event brokers in an efficient manner. We will first discuss *blanket capabilities* and then *group subjects*.

Blanket Capabilities

When the domain is granted access to a new resource, e.g. a newly deployed event type, the domain must propagate the new access rights to its event brokers. To simplify the process of propagating access rights to event brokers we can utilise a property of SPKI 5-tuple reduction:

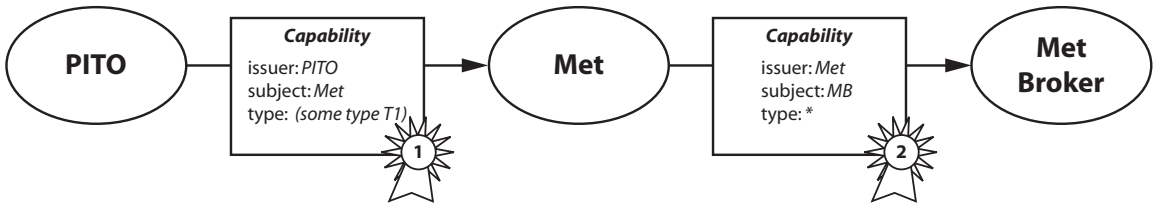

Figure 5.2: The blanket capability together with the capability issued to the Met domain authorises the broker to access type T_1.

in the 5-tuple reduction the authority of the reduced capabilities chain is the intersection of the authority fields of all the capabilities in the chain, as was explained in §5.1.1 and §2.6. Therefore the authority granted in the capabilities that are lower down in the chain, e.g. the capability issued to the Met Broker in Figure 5.2, is restricted by the capabilities higher up in the chain, e.g. the capability issued by PITO.

This property allows a domain to issue *blanket capabilities* to all of its event brokers. A blanket capability has an authority field that is a superset of the issuer's authority, but because of the 5-tuple reductions ends up being equal to the issuer's authority. A blanket capability allows the issuer to automatically delegate new access rights to all brokers without having to issue each one of them a new capability.

For example, in Figure 5.2 the Met Broker has been granted authority to access all event types by the Metropolitan Police domain. The domain on the other hand has only been granted access to the event type T_1. The authority granted to the Met Broker is the union of the authority fields in the two capabilities, C_1 and C_2, forming the certificate chain linking the broker to the type owners, i.e. if $A_1 = T_1 \wedge A_2 = \top \wedge S_1 = I_2$ then $A_1 \cap A_2 \implies T_1 \cap \top \implies T_1 = A_1$, where $\top$ signifies all possible access rights. Therefore the event broker's authority is always limited by the authority of the domain.

In the case where a domain ACS has been granted new access rights, e.g. access to the type T_2 in Figure 5.3, the blanket capability allows the Metropolitan Police domain to delegate the new authority to all event brokers in the domain simply by delivering the new capability to all the domain's event brokers. Once an event broker has obtained the new capability and is able to show it together with its blanket capability to a verifier, it has effectively been granted the access rights specified in the new capability. Without blanket capabilities the ACS would have to issue a new capability to each event broker in the domain each time the ACS was granted new access rights.

The access control policy cannot be compromised by the new capability leaking to an unauthorised principal in the same domain unless that principal has a blanket capability issued by the Metropolitan Police domain where the intersection of the authority fields is non-empty, i.e. $S_{\text{Met}} = I_{\text{Bob}} \wedge A_{\text{Met}} \cap A_{\text{Bob}} \neq \emptyset$. Therefore, assuming that the domain is able to enforce its own access control policy and avoid issuing blanket capabilities to unauthorised principals,

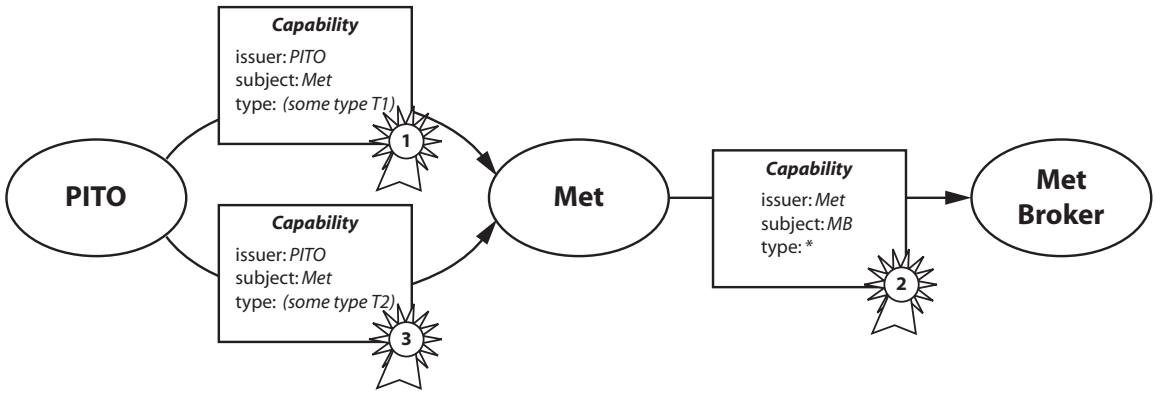

Figure 5.3: The blanket capability together with the new capability issued to the Met domain authorises the broker to access type T_2.

broadcasting the new capability in the domain does not present a security risk.

Similarly, leaking the new capability outside of the domain cannot compromise the access control policy, because the capability issued to the domain and the newly deployed capability must form a capability chain together, i.e. $S_{\mathrm{Met}} = I_{\mathrm{Bob}}$. Therefore, the new capability cannot be used with capabilities issued to other domains.

Instead of issuing blanket capabilities that grant event brokers access to everything that the domain can access without any restrictions, the blanket capabilities should be restricted, for example, by resource type or owner. In Figure 5.3 the blanket capability issued to the Met Broker grants access to all event types that the Metropolitan Police domain has access to. We would expect blanket capabilities to be restricted, for example, to one type of resources, i.e. networks, e.g. `network:*`, or event types, e.g. `type:*`, or alternatively to certain network or type owners, e.g. `network:` [P_{PITO}, *] or `type:` [P_{PITO}, *, *].

Group Subjects

Another alternative for distributing new access rights to event brokers is to use *group subjects* (We discussed group subjects in §2.6.3). In SPKI an authorisation certificate can be issued to a name rather than a principal. A principal is then linked to that name by issuing it a *name certificate*.

Each name is effectively a group, i.e. any number of name certificates can link principals to the same name. This allows for the creation of groups of subjects. In many cases where SPKI name certificates are used the group has only one member, i.e. there is a one to one mapping from a name to a principal. But this does not mean that other members cannot be added to the group at a later date.

By issuing all event brokers with a name certificate for a given name, e.g. *Met domain event brokers*, the domain can create an event broker group. After that the domain is able to issue authorisation certificates for that group, effectively delegating the given access rights to

all group members at the same time. When delegating rights to the group, the domain needs to issue only one certificate, as was the case with blanket capabilities.

During credential verification the verifier will first check that the event broker is a member of the given group, i.e. that the event broker has a name certificate linking it to the group name. Once the link between the principal and the name has been established, the verifier will check that the authorisation certificate is valid and grants the holder the required access rights.

In effect this approach is very similar to the blanket capabilities described in the previous section: both approaches allow the ACS to delegate access rights to a set of brokers quickly and efficiently without having to issue a new capability to each event broker.

The difference between the two approaches lies in their flexibility. With blanket capabilities the ACS must decide when issuing the blanket capabilities what kind of access rights the blanket capability should cover. For example, the ACS can restrict the blanket capability to a specific type owner. If later on the ACS wants to delegate access rights to the latest type from that type owner to a subset of the principals, the ACS must first revoke all the existing blanket capabilities and then issue new ones which exclude the new event type. In contrast, with group subjects the ACS is able to create a new group of those principals that should be delegated the new access rights and then delegate the access rights only to that group.

When delegating access rights via group subjects, the ACS can issue the authorisation certificates with relatively long life times while issuing short-lived name certificates to all group members. This way the ACS avoids having to re-issue the actual authorisation certificate, but it will still be able to easily control the group memberships of individual principals. Also, it is possible to issue name certificates with longer lifetimes to more trustworthy principals.

Group subjects resemble roles in *role-based access control* (RBAC) where access rights are granted to the role rather than directly to subjects, and access is controlled by controlling which subjects are able to enter a given role. Similarly, with group subjects access rights are granted to a group and group membership is controlled by issuing name certificates to principals.

The ACS is able to implement local policy management with RBAC either by always issuing a set of authorisation certificates to a subject when it enters a given role, or alternatively, with group subjects, by issuing a name certificate to the principal while maintaining long-lived authorisation certificates issued to the group.

5.1.4 Authorising Sub-Domains

Large domains might find it beneficial if they were able to create sub-domains to match their own organisational structure. In a nested setting the top-level domain delegates access rights to its sub-domains in the same way as resource owners grant access rights to top-level domains. Notice that sub-domains can be authorised by the resource owner directly, but in most cases we would expect the authorisation to follow the domain hierarchy, because domain internal structures should not be visible outside of the domain. For the sake of argument we assume

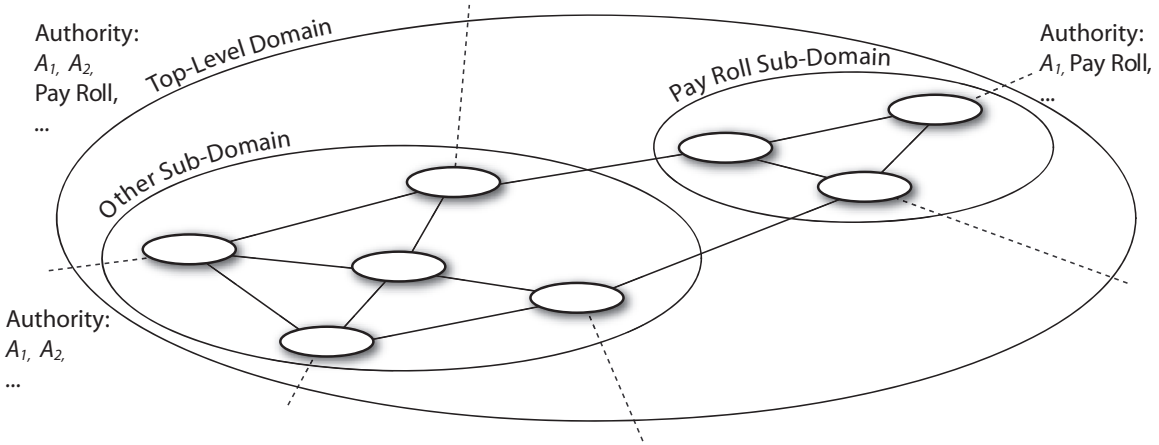

Figure 5.4: An enclosing domain can group more privileged brokers and event clients into their own privileged sub-domains.

here that sub-domains will not be directly authorised by the resource owner.

Sub-domains allow an organisation to implement a domain hierarchy consisting of multiple levels of sub-domains, each with their own set of privileges. For example, an organisation might want to restrict events related to employee salaries to a few trusted event brokers that form the Pay Roll domain and that are administered by a trusted system administrator. We assume that system administrators are able to circumvent an access control policy by logging on to an event broker and looking at the raw event flow. Therefore, we can assume that a system administrator is always able to access all events flowing through a broker administered by her. By placing the privileged brokers into a domain of their own and by granting only a trusted system administrator access to that event broker, we can prevent the untrusted system administrators from reading specific event types, i.e. salary events in this case.

Assuming that a sub-domain's authority is always a subset of its parent domain's authority, then the top-level domain's authority will always be the superset of all the authorities of all its sub-domains. If we were to place the Pay Roll brokers into their own Pay Roll sub-domain and the authority delegated to the top-level brokers was the same as the top-level domain's authority, as was suggested in §5.1.3, then the top-level brokers would always be authorised to access all events that were accessible to the top-level domain, including the confidential salary events. This suggests that either sub-domains must be more privileged than their parent domain, which would turn the domain hierarchy tree upside down, or alternatively the event brokers' authority cannot reflect the authority of theirs domain, which would go against the notion of event brokers representing their domain in the publish/subscribe system and therefore sharing the domain's access rights.

We can resolve this issue more elegantly by creating one sub-domain for the privileged event brokers and another one for the unprivileged event brokers. The privileged sub-domain is then granted access to the salary events while the unprivileged sub-domain is not. By placing

the event brokers into two sub-domains we can issue capabilities to the event brokers that allow them to share the privileges of their respective domains, as is shown in Figure 5.4. This approach maintains the subset relation between a sub-domain and its parent domain in the domain tree and we can also allow the event brokers to share all the access rights granted to their domain.

A domain is either a *domain group* in which case it contains only sub-domains, or it is a *broker group* in which case it contains only event brokers. A domain can never contain both event brokers and sub-domains. This definition is recursive allowing sub-domains inside a domain to contain again either other sub-domains or event brokers, but not both.

5.2 Resources and Access Rights

A publish/subscribe system has two types of resources that need to be protected: the event service and the event types. The event service is owned by a coordinating principal and the event types are owned by their respective type owners.

The access rights can also be divided into two groups based on the types of principals. Both event brokers and event clients need to be able to connect to the event service and to access the publish/subscribe API for specific event types. Type managers on the other hand need to be able to install event types on the event service and manage existing types.

The following sections discuss all the access rights related to both types of resources and to all three types of principals in more detail.

5.2.1 Event Service Access Rights

Principals need access to the event service in two cases: (i) when a principal wants to access the publish/subscribe API for any event type and (ii) when a principal wants to deploy a new event type, or a new version of an existing event type, on the event service. Here we equate the event routing performed by event brokers to accessing the event service, i.e. an event broker needs the right to access the event service in order to be able to join the broker network and route events. Both types of access rights are granted by the coordinating principal, that is seen as the owner of the event service.

Connecting to the Event Service

All event clients and brokers need to be authorised to connect to the publish/subscribe system in order to be able to access it. This provides network layer access control and prevents unauthorised parties from connecting to the event service and seeing the system's internal traffic. If a malicious node is able to access the event service, it is able to launch a simple *denial of service* (DoS) attack by issuing a large number of subscription requests, or by injecting invalid routing

messages to the broker network, which can lead to network partitions. We can easily protect against such trivial DoS attacks by controlling access to the broker network.

In our example below, PITO, as the coordinating principal, invites another domain, the Metropolitan Police, to join the shared publish/subscribe infrastructure by issuing a capability to the Metropolitan Police domain that grants it the right to connect to the given publish/subscribe network. The capability specifies the name of the network and the authorised action, [P_{PITO}, `UK Police Network`] and *connect*, respectively:

Name	Value
issuer	P_{PITO}
subject	P_{Met}
network	[P_{PITO}, `UK Police Network`]
action	connect

The Metropolitan Police domain delegates the right to connect to the UK Police Network to all of its event brokers. This allows the domain's publish/subscribe infrastructure, i.e. the event brokers, to join the shared broker network. Here we specify a capability that grants one of the event brokers in the Met domain access to a specific publish/subscribe network. Notice that the capability has been issued to the group `Met Brokers` rather than to a single broker.

Name	Value
issuer	P_{Met}
subject	`Met Brokers`
network	[P_{PITO}, `UK Police Network`]
action	connect

Finally those event clients in the Metropolitan Police domain that need access to the UK Police Network, i.e. Detective Smith in this case, are issued capabilities that authorise them to connect to the broker network:

Name	Value
issuer	P_{Met}
subject	P_{Smith}
network	[P_{PITO}, `UK Police Network`]
action	connect

Together the three capabilities form a certificate tree that connects the event broker and the detective to PITO, as seen in Figure 5.5. The 5-tuple reduction operation collapses one branch of the capabilities tree into a single capability where the issuer is the resource owner, the subject is the leaf of the tree, e.g. Detective Smith or Met Broker, and the validity and authority fields are the intersection of all the validity and authority fields in all the capabilities in the tree branch, respectively. Notice that the Met Broker's capability chain includes a name certificate that is not shown in the figure. The name certificate binds the Met Broker's identity, P_{MB}, to the `Met Broker` group.

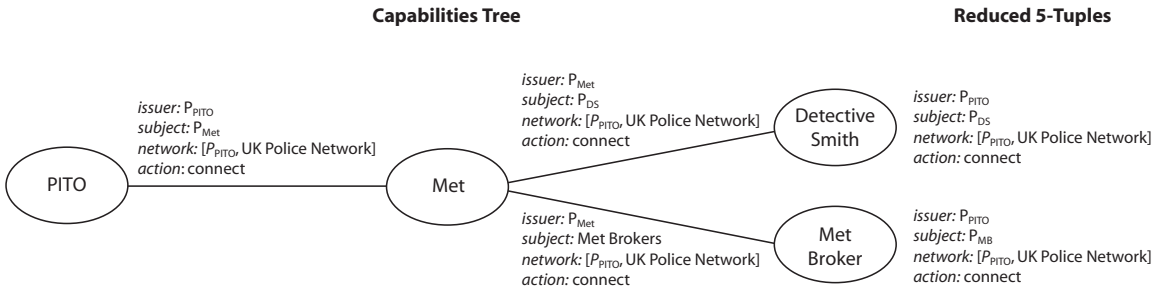

Figure 5.5: The principals and the capabilities form a tree where the principals are nodes and the capabilities are vertexes.

Installing Event Types

The other type of access right issued by the coordinating principal and relating to the event service as a resource is the right to install event types on the event service. This right is relevant to principals in the system that want to either create new event types or deploy new versions of existing event types. Again the coordinating principal issues a capability to a domain in order to authorise the domain to issue new event types on the system:

Name	Value
issuer	P_{PITO}
subject	P_{CCS}
network	[P_{PITO}, `UK Police Network`]
action	install

The domain then delegates this authority to those members of the domain who should be authorised to deploy new event types or updated versions of existing event types. In this case the CCS domain has granted the install right to the Billing Office:

Name	Value
issuer	P_{CCS}
subject	P_{BO}
network	[P_{PITO}, `UK Police Network`]
action	install

Notice that the principal deploying the event type does not need to connect to the event service. The principal simply defines an event type, signs it, and stores it in a type registry. An event client will then pick up the type definition from the registry and present the definition to its local broker as part of a publish/subscribe request. Therefore, a principal deploying new event types does not need the *connect* access right in order to be able to deploy event types.

5.2.2 Event Type Access Rights

There are two cases where a principal needs access to an event type:

i. when accessing the publish/subscribe API for a given event type, i.e. publishing or subscribing to an event type, and

ii. when updating an existing event type definition and releasing the modified definition as a new version of the existing event type.

In the first case the principal will also need the *connect* right to the appropriate publish/subscribe network for using the publish/subscribe API. In the latter case the principal will need the *install* right to be able to install a valid event type definition on the event service.

Both access rights described in this section specify the event type that they apply to. The type field specifies a 3-tuple pattern of the type name and therefore it can be used to match a specific version of an event type or all the types sharing the same prefix. For example, [P_{PITO}, `uk.gov.pito.Numberplate`, 1234] will match the version 1234 of the *Numberplate* event type whereas [P_{PITO}, *, *] will match all types owned by PITO.

Both the identity and version fields either match one specific identity or version or all identities and versions. Because the version numbers are not guaranteed to be linear, as discussed in §4.3.1, we cannot use relational operators in defining patterns that match all versions below or above a given version number. The name field is treated as a character string, which allows the pattern to include wildcards for prefix, postfix and infix matching.

In some cases it is nonsensical to specify a type field that matches more than one event type. For example, granting a principal *publish* or *subscribe* rights to all event types from a specific type owner while specifying the set of attributes that she can access does not make sense, but is still a valid capability.

For the *manage* right the type field can match any set of event types. We expect that in most cases specifying the version number of an event type is not desirable, because it prevents the event clients from using newer event types and therefore forces the type owner to support the older event type indefinitely.

Accessing the Publish/Subscribe API

From an application's point of view the right to access the publish/subscribe API for a specific event type is the only access right that really matters. Access to the API allows the client to publish or subscribe to a specific event type. With the right to connect to the event service, but without the right to publish or subscribe, the application is unable to utilise the event service and is therefore unable to communicate with other nodes over the publish/subscribe system.

A publish/subscribe API request is always specific to an event type, i.e. a client advertises or subscribes to a specific event type instead of making general requests. The owner of the event

type is responsible for delegating API access rights to domains on a per type basis. The type owner issues domains capabilities that specify the event type and the requests that the domain is allowed to make concerning that type (i.e. publish or subscribe). Note, that the *publish* right is checked at advertisement time rather than at publication time. This allows the system to verify the authority once in the beginning of a session before any routing state has been created in the broker network, instead of having to verify the authority for each published event.

In the following example PITO, as the type owner, has granted the Congestion Control Service both the right to publish and the right to subscribe to events of any version of the *Numberplate* event type. The capability grants the CCS domain access to all attributes in the event type.

Name	Value
issuer	P_{PITO}
subject	P_{CCS}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, `*`]
action	publish
	subscribe
attributes	*

The CCS domain is responsible for further delegating a subset of these access rights to the members of the CCS domain. The CCS domain has granted the Billing Office the right to subscribe to *Numberplate* events and to read the `numberplate` and `timestamp` attributes shown below:

Name	Value
issuer	P_{CCS}
subject	P_{BO}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, `*`]
action	subscribe
attributes	numberplate
	timestamp

In reality the attributes would be identified by their UIDs rather than their names in order to allow the capability to be valid even when the attributes have been renamed (See §4.3.2 for more details on renaming attributes). We will use attribute names instead of UIDs in our examples in order to maintain readability. In an implementation the attribute names can be included in annotation attached to the attributes in order to provide better user experience in type editing tools.

The following table shows a capability issued to a CCTV camera placed near Victoria station that is granted the right to publish *Numberplate* events from that location:

Name	Value
issuer	P_{CCS}
subject	P_{CCTV}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, *]
action	publish
attributes	location = `Victoria`
	numberplate
	timestamp

The capabilities specify the access rights granted to an event client at the granularity of individual attributes. Each attribute specified in the capability is accessible to the principal for the specified action, e.g. the CCTV camera at Victoria is allowed to set the value for the `timestamp` field when it publishes a *Numberplate* event. Note that the attributes do not have to exist in the event type. This allows a capability to apply to updated versions of the event type where attributes have been removed. Similarly the capability restricts the event client's access to those attributes that are mentioned in the capability and therefore the event client is unable to access attributes that have been added to a new version of the event type.

Specifying the attributes in the capability allows the system to limit attribute visibility in cases where the event client does not need to see all of the event content. For example, in the Congestion Control scenario the Billing Office needs access to the numberplate of a car entering the congestion controlled area, but there is no need for it to access the location where the car was sighted, because the congestion fee based on the vehicle entering the area, not on its location when there:

Name	Value
issuer	P_{CCS}
subject	P_{BO}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, *]
action	subscribe
attributes	numberplate
	timestamp

Similarly the Statistics Office needs to see timestamps and locations for each numberplate sighting, but it has no need to know the specific numberplates in order to collect traffic statistics for a given area:

Name	Value
issuer	P_{CCS}
subject	P_{SO}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, `*`]
action	subscribe
attributes	location
	timestamp

Detective Smith on the other hand must see all attributes to be able to track down a vehicle, but her subscription includes a filter on the numberplate attribute in order to prevent the detective tracking arbitrary vehicles:

Name	Value
issuer	P_{Met}
subject	P_{Smith}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, `*`]
action	subscribe
attributes	location
	numberplate = `AE05 XYZ`
	timestamp

Unlike domains and event brokers, event clients can be issued capabilities that specify restrictions on event attribute values. For example, the value of the `location` attribute in the capability issued to the CCTV camera is forced to the value `Victoria`. For the CCTV camera this means that its local broker forces the value of the `location` attribute to `Victoria` regardless of the value the CCTV camera has set to that attribute. Similar restrictions can also be used when granting subscription rights to an event client. For example, Detective Smith of the Metropolitan Police has the right to subscribe to *Numberplate* events where the value of the `numberplate` attribute is `AE05 XYZ`. Detective Smith's local broker is responsible for enforcing the restriction by adding an appropriate filter expression to Detective Smith's subscription.

It is impossible for type owners or domains to specify similar attribute level restrictions for domains, because there is no way for the issuer to enforce those restrictions. A domain on the other hand is able to enforce the restrictions on event clients, because an event client accesses the event service by connecting to an event broker that is a member of the same domain and acts on the ACS's behalf.

When a client has access only to a subset of the attributes in an event type, the client hosting broker will replace the other attributes with `null` values. In the case of publishers, the PHB sets all attributes that the publisher is not authorised to access to `null`. If a subscriber is not authorised to access an attribute in a publication, the SHB delivers the publication to the subscriber with the inaccessible attributes set to `null`.

We expect that in most cases publishers would have access to all attributes, albeit some attributes might have forced values as is the case with the CCTV camera above. Subscribers on the other hand might be more restricted with respect to access to attributes. We imagine that in many cases where a number of subscribers with different roles access the same events the subscribers would be granted access to different sets of attributes as is the case in our example application with the Billing Office, Statistics Office, and Detective Smith as subscribers. Forced subscription filters would probably be used quite frequently in cases where the subscriber is only authorised to access a subset of the event stream. Such cases would include our example with Detective Smith as well as any case where the subscriber is charged for accessing the event stream as is the case in the Stock Ticker example from §1.2.1.

Type Management

We assume that event type definitions need to be revised during their lifetime, either because the original definition was incorrect or because the application requirements have changed since the event definition was deployed. Type management is the duty of the type owner, but if necessary she can delegate type management duties to other principals by issuing them a capability with the *manage* access right:

Name	Value
issuer	P_{PITO}
subject	P_{TM}
type	[P_{PITO}, `uk.gov.pito.Numberplate`, `*`]
action	manage

This capability allows the type manager (P_{TM}), whose job in the PITO organisation includes deploying new versions of PITO owned event type, to issue new versions of the *Numberplate* event type.

Delegating type management duties to other principals allows type owners to spread the responsibility of managing an event type to other principals in cases where the type owner is either unable to perform their duties at this time (e.g. they are leaving the organisation) or when the type owner wants to distribute the load of type management between multiple principals.

We can subdivide the *manage* right into specific management operations, as described in §4.3:

i. add attribute,

ii. remove attribute,

iii. rename attribute,

iv. change attribute type.

Fine grained access rights give a type manager more control when delegating type management duties to other principals. For example, a type manager could delegate the right to add attributes to a less trusted principal so that they could add new attributes to an event type if necessary, but are prevented from removing vital attributes from the event type and thereby breaking existing applications.

Unfortunately verifying that the type manager has acted within its authority requires that the verifier is able to compare this version of the event type to the previous version. The previous version of the event type could be incorporated in the new type definition, but this will result in larger event type definitions. Alternatively the event type definition can reference the previous version by name and it is up to the verifier or the prover to lookup the previous version from a type registry. By comparing the two versions with each other the verifier can ascertain that the changes made to the old version are indeed authorised by the type manager's credentials.

We feel that in most cases the extra granularity in granting type management rights to type managers is not necessary and thus it is simpler to just grant the *manage* authority rather than the right for individual type management operations. That is, if a type manager is not trusted to manage the type definition with respect to all the type management operations specified above, then the type manager should not be authorised to manage the type at all.

Note that it does not make sense to include the version number in the type field of the capability unless we support the finer granularity type management operations, i.e. restrict the type manager's access right to, for example, renaming existing attributes. If a type manager has been granted to *manage* right, it does not make a difference what the original version of the event type is, the type manager is able to change any it to any other version of the same event type by using the above mentioned operations.

As in the case of the *install* right, the credentials granting the principal the *manage* right need to be embedded in the type definition so that the broker verifying the validity of the type definition is able to verify the type manager's right to deploy a new version of the given event type. Note that the type manager must also have the *install* right in order to be able to deploy updated event type definitions on the publish/subscribe system.

We would expect types to be managed and created by the domain's ACS or alternatively a specific principal responsible for event types in that domain. In either case the same principal will most likely be responsible for both managing the type definition and access rights to that type across the type's lifespan.

5.3 Verifying Authority

In distributed, capability-based access control systems a principal's authority is usually verified by executing an interactive protocol between the principal and the verifier. During the protocol (i) the principal authenticates herself to the verifier, (ii) presents her credentials to the verifier,

and (iii) issues an access request concerning a specific object. The verifier confirms the principal's identity, verifies the presented credentials, and finally, assuming that the credentials are valid, executes the principal's request.

5.3.1 Authentication

In SPKI the principal's identity is represented by a public key. The principal authenticates herself by proving that she owns the private key corresponding to the public key. A principal proves the ownership of a private key by signing a piece of information with the private key. The verifier is then able to verify the signature with the principal's the public key, thus verifying that the principal does indeed own the private key corresponding to her identity. The principal's public key is included as the subject in one or more of the provided credentials.

Public key cryptography allows for a number of identification protocols based on signing a *nonce*[1]. The verifier is able to ascertain the principal's identity by verifying the signature on the nonce with the principal's public key.

The ISO/IEC 9798-3 [ISO98] standard defines the following three mechanisms for authenticating a principal using digital signature techniques [MOV96]:

i. *unilateral authentication with timestamps:*

$$A \rightarrow B : t_A, B, S_A(t_A, B)$$

Principal A sends the authentication message to B, who verifies that the timestamp is valid, i.e. it is fresh with respect to some pre-defined grace period, that the identity B in the message is its own, and that the signature across these two values is correct.

ii. *unilateral authentication with random numbers:*

$$A \leftarrow B : r_B$$
$$A \rightarrow B : r_A, B, S_A(r_A, r_B, B)$$

Verifier B sends a challenge message to principal A. Upon receipt of the challenge principal A generates her own random number r_A and sends a reply back to B. B verifies that the identifier is its own and that the signature over the two random numbers and the identifier is correct. The random number r_A is used to prevent chosen-text attacks where A might be tricked into signing, for example, some legally binding document.

[1] A *nonce* is a number that is used only once for the same purpose. It is used to prevent the replay of messages. A nonce can be a timestamp, a random number, or a sequence number.

iii. *mutual authentication with random numbers:*

$$A \leftarrow B : r_B$$
$$A \rightarrow B : r_A, B, S_A(r_A, r_B, B)$$
$$A \leftarrow B : A, S_B(r_B, r_A, A)$$

The message verification in this protocol is the same as in (ii) above. This protocol, compared to the two previous ones, allows for simultaneous authentication of both parties, whereas the previous protocols authenticated only principal A.

Notice that often with identity certificates, e.g. X.509 identity certificates, the goal of executing the authentication protocol is to verify the principal's identity, i.e. that they own the presented identification certificate. Therefore, the identity certificate must be either provided to the verifier in the protocol messages or in some out-of-band mechanism. The certificate gives the verifier the public key to use to check the digital signature in the authentication message. In our case the goal is to prove the ownership of the key pair rather than the ownership of an identity certificate. The verifier still needs the public key to check the digital signature. The principal's public key (or its hash value) is the subject of the last authorisation certificate in the certificate chain. Therefore the public key is provided to the verifier as a part of the principal's request.

When authenticating the principal to the verifier, we could simply use the first method, *unilateral authentication with timestamps*, described above. This would allow us to piggy-back the authentication message with the credentials and the request as part of single message being sent to the verifier.

The second method, *unilateral authentication with random numbers*, is only useful if the two parties do not have access to an accurate time source that can be used as a source for nonces. The protocol includes an initial challenge message that can be avoided in the first protocol. In our environment we assume that all nodes have access to a time source that is accurate enough within a certain delta.

In MAIA we want both parties of a connection to mutually authenticate each other (we discuss the motivation for this in detail below in §5.3.3). We can achieve this in an ad-hoc fashion by executing either of the unilateral protocols twice, but that results in two independent runs of the protocol that cannot be logically associated with each other. By implementing the third protocol described above, *mutual authentication with random numbers*, both parties can authenticate themselves to each other as part of the same protocol run.

5.3.2 Authorisation

To prove her authority to access a specific event type the event client presents the verifier with a set of credentials that together provide evidence of the event client's authority. These credentials

include all the authorisation certificates that form the certificate chain between the principal and the type owner. If the any of the certificates in the chain includes group or named subjects, then the principal must also provide the necessary name certificates. Similarly with threshold subjects, the principal must provide authorisation certificates from k of the n issuers to prove delegation.

The verifier checks the validity of all the provided certificates by checking that the signature is correct, that the certificate has not expired and that the possible on-line tests pass (We discuss SPKI on-line tests in more detail in §6.2.3). The certificates are then mapped to tuples.

Names in the tuples are replaced with public keys or hashes of public keys by performing a 4-tuple reduction on the name 4-tuples.

The authorisation 5-tuples are then reduced to a single 5-tuple by recursively applying the 5-tuple reduction rule on consecutive pairs of 5-tuples.

Finally the verifier checks that the authority field in the reduced 5-tuple includes the requested operation on the specified object. The verifier then executes the principal's request.

There must be a clear link between the resource owner, i.e. the root principal in the capability chain, and the object being accessed so that the verifier can form a certificate loop from the chain and verify it. This link is formed by an ACL in SPKI. An ACL in SPKI specifies the owner of an object. The SPKI specification does not give a formal description of an ACL. The informal description of an ACL provided in the specification is simply an authorisation certificate body without the issuer field. In practise implementations and applications are free to choose their own ACL formats. In MAIA we use event type definitions as an ACL when dealing with event type access rights. Access rights related to the event service, e.g. the right to join the event service, are based on trusted principals. I.e. the owner of the network resource is specified explicitly in the participant's system configuration similarly to how X.509 root certificates are specified in web browsers.

5.3.3 Verification in MAIA

In MAIA the principals making access requests are event clients, event brokers, and type managers. The verifier is always an event broker that is connected to the broker network.

Interactive verification happens only between an event client and an event broker or alternatively between two event brokers. For type management rights the type manager's authority is verified at publish/subscribe request time when the event client presents an event type to an event broker as part of its advertise or subscribe request. This is because a type manager never interacts with the event brokers directly (the event client acts as the type manager's proxy).

In order to present its credentials to the verifier the type manager embeds them in the type definition that she has signed. The type manager's signature on the type definition, along with the type manager's identity in the name tuple, authenticate the type manager to the verifier even though the two never communicate with each other directly. The verifier is able to trust

the presented signature and does not have to worry about replay attacks, because the signed document, i.e. the type definition, is a self-contained access request that specifies the requested operation (deploy event type) and the related data (event type definition). This is similar to digitally signing an email and sending it to the recipient.

With API related access rights the principal always connects to the verifier directly and is therefore able to present its credentials to the verifier in an interactive session. The connecting principal presents her credentials to the verifier in order to prove that it is authorised to make the given API request.

Following the principle of least privilege, the event client can decide later in the session to disclose more credentials if it wants to make other publish/subscribe requests that are not covered by the already verified credentials.

When an event broker connects to another broker, it must also verify what the other broker's credentials are so that it can decide whether it can deliver plain-text events to the broker or if it must first encrypt them in order to prevent the unauthorised broker from accessing the event content. We will discuss event encryption in more detail in Chapter 7. Notice that while an event client can choose to disclose only those credentials that authorise it to perform a specific publish/subscribe request, the event brokers must disclose all their credentials so that (i) an event client can decide if the broker has the required authority to implement all future requests for the client and (ii) a neighbouring event broker can decide which events it can deliver in plain-text and which ones it needs to encrypt (We discuss in §7.4 the possibility of avoiding encryption operations when the event broker knows that the receiving event broker has the same access rights to the event.).

The brokers are able to increase their verification performance by caching the verification results of single capabilities between sessions and storing the capabilities on disk. Caching verification results will allow the broker to check the cache for a previous verification result before committing itself to an expensive verification procedure. Storing capabilities on disk will allow the broker to read and verify those capabilities as part of its bootstrapping process, thus performing the expensive verification procedure in advance rather than during normal runtime when it is also responsible for executing client requests and routing events. We provided performance measurement results in §4.5 which showed how expensive capability verification was compared to using cached results.

5.4 Delegating Root Authority

Decentralised trust management is based on the idea that the owner of a particular resource is the ultimate authority, with respect to access control policy and decision making, and is therefore responsible for managing the access control policy for that resource. Nevertheless the resource owner is able to delegate this root authority to another principal by granting the

principal a capability that grants her all access rights related to the resource. Typically we would expect a capability to have certain validity conditions and, if nothing else, at least an expiration date. By issuing the above described capability to a principal without any validity conditions the resource owner is effectively delegating root authority, i.e. all access rights, to another principal. The capability allows the new resource manager to manage access to the resource as if it was the original resource owner, as long as it always includes evidence that it has been delegated authority to manage access to the resource, i.e. the capability issued by the original resource owner.

The resource owner is still able to issue capabilities related to the resource herself, but it has no way of revoking the other principal's authority. This mechanism allows resource owners to permanently delegate resource management duties to other principals in the system. For example, a domain, that is about to leave the system, might delegate type management duties permanently to another principal in the same domain in order to allow other domains to carry on using its event types.

With event type definitions the event type name and the signature must be linked together, as explained in Chapter 4, therefore the capability delegating root authority to a new type manager must be included in the type definition. Including the capability in the type definition will allow anyone to verify the authenticity and integrity of the type definition. With access rights to the publish/subscribe network or to the publish/subscribe API for a given event, the capability used to delegate the root authority must be included in the certificate chain that the client shows to the event broker verifying the authority.

The use of threshold subjects as type owners and network owners simplifies the management of that resource significantly, because the principals included as members of the threshold subject can be changed as described in §2.6.4. Therefore, when threshold subjects are used to create publish/subscribe networks or event types, there is no need to delegate root authority to another principal at all.

5.5 Access Control in Topic-Based Publish/Subscribe

While most of this chapter is concerned with event types, the same principles are equally applicable to topic-based publish/subscribe systems. For example, topic-based publish/subscribe systems share the two fundamental access rights described in §5.2, namely the right to connect to the event service and the right to access the publish/subscribe API.

The right to create new topics should also be controlled by the coordinating principal as is the case with installing new event types in our type-based system.

For a broker to verify the authority of a type owner to install a new event type the broker needs to see the type owner's credentials. In our type-based system, as described earlier in this chapter, the credentials are embedded in the type definition which allows any principal in the

system, that has access to the type definition, to verify the type manager's authority to deploy that type definition. With topic-based publish/subscribe there exists nothing similar to type definition that could be used as a container for the topic owner's credentials.

In order to allow the topic owner to present her credentials to the publish/subscribe system we could devise a topic definition that would embed the credentials similarly to our type-based system, but it would be difficult to justify why the event clients should be interested in the topic definitions and provide them for the local broker, as is the case with type definitions (See §4.4.1).

Instead of forcing the event clients to provide the topic definition for the local brokers, we could implement a topic registry in the event service. A topic owner would create a new topic by passing a topic definition for a local broker which would verify the topic owner's authority and store the topic definition in the event service. A topic definition would contain the name of the topic, the owner's identity, the credentials authorising the topic owner to create the topic, and a digital signature that protects all the above fields. Other brokers could then lookup the topic definition from the registry when they are issued a request related to that topic. It is important that all event brokers in the system are able to verify the topic definition for themselves, otherwise a malicious event client and a malicious event broker from the same domain could collude and create an unverified topic definition. Remember that the topic's name tuple must contain the topic owner's identity as was described in §4.6.

In a hierarchical topic-based publish/subscribe system, a publisher publishing events under a topic must have publication rights to all of the super-topics as well in order to be authorised to publish the events. Whereas for a subscriber it is enough to be authorised to subscribe to the current topic only, with no access rights with respect to the inherited topics.

5.6 Related Work

Zhao and Sturman propose an approach to dynamic access control in a content-based publish/subscribe system in [ZS06]. In contrast to our work they propose a centralised access control list based architecture, which, while perfectly acceptable for single domain deployments, will not effectively scale to multiple domains. The proposed scheme maintains a central ACL in the system, which is consulted when event clients make new advertisement or subscription requests. The ACL server becomes a bottle-neck in the system when the number of authorisation requests increase. Also, when the centralised policy is changed, the policy changes are pushed to all interested brokers, i.e. brokers hosting event clients that are affected by the policy change. This approach provides a very fast revocation mechanism, but it also results in a large number of messages being sent immediately after the policy change. Our approach to revocation will be discussed in Chapter 6.

Zoltán Miklós proposes an access control mechanism for content-based publish/subscribe

systems in [Mik02]. The paper treats credentials as subscription or publication filters. That is, a credential defines what event content a principal is allowed to publish or subscribe to as a filter on that event content. The principal is allowed to setup advertisements and subscription that are covered by the credential. Miklós' proposal is very similar to how we suggest to restrict publishing and advertisement rights based on the event content, as discussed in §5.2.2. However, our proposal is less formal and would benefit from Miklós' more formal treatment. From a practical point of view, Miklós does not describe how access rights are granted to a principal, or how the event broker authorising the event client's request is made aware of the current policy. These issues have been the main concern of our work, so we can envision integrating Miklós' work with our own approach.

As a precursor to our work, Belokosztolszki et al. presented an RBAC-based access control architecture for publish/subscribe systems [BEP+03]. We have expanded on that work by decoupling the RBAC policy management system from the access control verification mechanism. That is, our architecture allows the use of RBAC within domains, but uses SPKI authorisation certificates as credentials between the domain members and the event brokers.

We base our access control model on that presented in a number of papers relating to the topic: [BEMP05] introduces the multi-domain environment and proposes a high-level access control approach based on role-based access control, and [PB05], as discussed in the previous chapter, introduces *secure event types* and secure names. Finally, the access control architecture proposed in this chapter is based on the work originally published in [PEB06] and [PEB07].

5.7 Summary

We have presented a capability-based access control architecture for multi-domain publish/subscribe systems. By applying decentralised trust management principals, we are able to administer and enforce access control in publish/subscribe systems that span multiple independent administrative domains both in a convenient and scalable manner. While the chapter concentrates on type-based publish/subscribe systems, the presented ideas are equally applicable to topic-based and content-based publish/subscribe systems.

We have identified two resource types, broker networks and event types, that are present in a type-based publish/subscribe system. And we have identified five operations, *connect*, *install*, *publish*, *subscribe*, and *manage*, that can be executed on one of the two resources. Our proposed access control model allows us to control access to both types of resources and to authorise principals to execute all five operations independently.

The proposed architecture is multi-tiered, i.e. resource owners authorise domains rather than event clients and brokers. The domain is then responsible for further delegating that access right to domain members. We see event brokers as representing the domain in the publish/subscribe system and therefore they are typically delegated all of the domain's authority. The multi-tiered

architecture also seamlessly supports a hierarchy of domains. This allows large domains to divide their organisation into a number of sub-domains internally.

The presented access control model provides access control at the edges of the broker network. That is, the brokers are responsible for enforcing a decentralised access control policy that is defined in the form of signed capabilities. We provide a mechanism for controlling which brokers can join the broker network, but once a broker has been allowed to join the publish/subscribe system it is able to implement any publish or subscription request without any restrictions.

This approach works well if we can assume that the brokers are trustworthy. While this assumption can be valid inside a single domain, it most certainly does not hold across domain boundaries. Therefore enforcing access control inside the broker network becomes an issue. We address this in the following chapter by encrypting the content of events while they are in transit in the broker network, thus preventing unauthorised brokers from decrypting the events. Effectively we move access control enforcement from the brokers to the key servers managing the encryption keys, and thus evolve away from an access control approach that relies on trustworthy brokers.

CHAPTER 6

Policy Management

The dissertation has thus far focused solely on how to implement enforcement of access control in a multi-domain environment in a scalable and manageable manner. We have not touched on policy management or credential revocation in previous chapters. While the dissertation concentrates on the mechanisms for enforcing an access control policy rather than the management of that policy, we feel that for the sake of completeness we must discuss the options that are available for resource owners and domains to implement access control policy management and credential revocation.

The remainder of this chapter will address a number of practical issues relating to policy management in a real-world deployment of MAIA. In §6.1 we provide an illustration of how the *Open Architecture for Secure Interworking Service* (OASIS) system can be used to manage policy in a domain. One of the more serious disadvantages of capabilities is the fact that it is non-trivial for an issuer to revoke a capability that has already been delivered to a principal. We discuss credential revocation in §6.2. In §6.3 we present a mechanism for delivering SPKI validity statements to event brokers over the publish/subscribe infrastructure. Related to the revocation of capabilities, we discuss in §6.4 the possibility of delegating part of the dynamic policy evaluation to the event brokers as a means of improving the efficiency of the system. In §6.5 we discuss how capabilities can be delivered to principals. Finally we present related work in §6.6 and a summary of the chapter in §6.7.

6.1 OASIS

We stated earlier in Chapter 5 that in MAIA the policy management at the resource owner and the domains is decoupled from the enforcement of that policy at the event brokers. We also

wrote that both the resource owners and the domains were free to implement a policy management approach of their choice without affecting the enforcement of that policy in the multi-domain environment. For example, a resource owner might consult a simple *access control list* when issuing capabilities to domains. Similarly a domain might use a more sophisticated policy management system, for example *Ponder*[DDLS01] or *OASIS*. We will use OASIS as an example in this section.

The *Open Architecture for Secure Interworking Services* (OASIS) [BMY02, BMY03] is an established, distributed, role-based access control (RBAC) [FK92, SCFY96] system that can be used to manage access control policy for a domain or a type owner. It provides a comprehensive rule-based means to specify roles and access rights associated with a given role. Principals acquire role memberships and activate access rights associated with a role according to role activation and authorisation policies, respectively.

In OASIS a principal acquires the privileges that authorise it to access an object in two steps: first, the principal acquires membership of a role by activating a role membership rule. Second, the principal activates an authorisation rule that results in the principal, as a member of a role, to be issued with the appropriate capabilities.

A role activation policy comprises a set of rules, where a role activation rule for a role r takes the following form:

$$r_1, \ldots, r_n, a_1, \ldots, a_m, e_1, \ldots, e_l \vdash r$$

where r_i are prerequisite roles, a_i are appointment certificates and e_i are environmental constraints. The appointment certificates represent persistent credentials, e.g. a physician's medical license, an employment contract between an employee and an employer, or in Detective Smith's case a court order authorising her to track a given numberplate. The environmental constraints allow restrictions to be imposed on when and where roles can be activated, e.g. role activation can be limited to a physician's working hours, or in Detective Smith's case, as we shall see below, to her being assigned to the investigation involving the tracking of a given numberplate.

A predicate that must remain true for the principal to remain active in the role can be tagged as a *role membership condition*. Such predicates are monitored, and their violation triggers revocation of the role membership and related privileges from the principal. We discuss in §6.4 how the evaluation of such predicates could be delegated to the verifier rather than being implemented at the issuer, i.e. at the ACS.

An authorisation rule for some privilege p takes the form:

$$r, e_1, \ldots, e_l \vdash p$$

where r is an active role, e_i are environmental constraints, and p is a privilege granted to the principal. An authorisation rule takes only one role as input, but a variable number of environmental predicates, and results in a single privilege being issued to the principal. An authorisation policy comprises a set of such rules. Note that OASIS has no negative rules, and satisfying any one rule indicates success.

OASIS roles and rules can be parameterised. This allows fine-grained policy requirements to be expressed and enforced, such as exclusion of individuals and relationships between them. Without parameterisation it becomes necessary to define an unmanageable number of roles for larger systems. For example, the parameterised role *npTracker(numberplate)* allows the same *npTracker* role to be used in all investigations that require access to the numberplate tracking system. Parameterising the role binds the role to the specific numberplate thereby preventing the detective from abusing the granted privilege by tracking arbitrary numberplates. Without parameterisation it would be difficult if not impossible to express the access control policy at such a fine level of detail.

6.1.1 OASIS Policy in Our Example Scenario

In our Congestion Control example Detective Smith is only permitted to receive events relating to the sighting of a particular numberplate. We have indicated how the publish/subscribe system can enforce these types of access control rules, but have not discussed how to specify this in terms of policy within a domain.

In this section we show how the OASIS policy language could be used to specify the simple rules required by our example scenario. We propose that the courts are equipped with a means to issue a warrant as an OASIS appointment certificate. This appointment certificate, represented by *courtOrder* below, has parameters that specify which case and which numberplate the warrant has been issued for. The appointment certificates are set to expire when the court order expires.

The predicate *detective* ensures that only detectives are able to activate the role membership rule. The constraint is parameterised with the principal's identity, which corresponds to an identity that can be verified when the principal tries to activate the role. This identity can be either an X.509 identity or a public key as used in SPKI to represent principals. If the predicate is implemented as an environmental constraint, it will probably result in a database lookup to check that the presented identity is a detective. Alternatively the predicate could be represented by an appointment certificate held by the principal that indicates that the principal is indeed a detective of the Metropolitan police.

The domain access control policy can finally ensure that role membership is granted only to detectives who have been assigned to the case. We use the $caseAssignment$ environmental predicate to record the mapping between cases and detectives.

An appropriate OASIS role activation rule for numberplate tracking in the Met domain would be:

$$courtOrder(\texttt{caseId}, \texttt{numberplate})$$
$$detective(\texttt{detectiveId}),$$
$$caseAssignment(\texttt{detectiveId}, \texttt{caseId}) \vdash npTracker(\texttt{numberplate})$$

This rule grants Detective Smith membership of the *npTracker* role. That is, the Detective is issued a *role membership certificate* indicating that she is a member of the role. The role is parameterised with the numberplate of the vehicle in question that is specified in the court order. The binding prevents the Detective from using the role membership to track arbitrary numberplates.

The role membership rule by itself does not grant the principal any access rights. The policy must specify one or more authorisation rules that allow the principal to acquire actual access rights. In our example the policy would include the below authorisation rule:

$$npTracker(\texttt{numberplate}) \vdash Numberplate.subscribe(location,$$
$$timestamp,$$
$$numberplate = \texttt{numberplate})$$

This rule grants the holder of an *npTracker* role membership certificate the right to subscribe to *Numberplate* events with a filter over the *numberplate* attribute. The filter value is the numberplate string that is included in the role membership certificate. The principal is granted the right to read the three attributes, (i.e. *location, timestamp,* and *numberplate*) of delivered events, i.e. the SHB delivers events to the subscriber with all three attributes intact.

We have used SPKI in MAIA to implement capabilities. If we were to implement policy management in a domain with OASIS, the above rule would have to result in an SPKI authorisation certificate that would grant the principal subscription rights to the Numberplate events, as described in §5.2.2. The OASIS implementation could either issue SPKI authorisation certificates directly, or the ACS could implement a level of indirection where a rule activation would first result in an OASIS capability that would then result in an SPKI authorisation certificate being issued to the principal.

6.2 Access Rights Revocation

Compared to access control lists, capabilities suffer from two distinct disadvantages. First, because the currently active access control policy is distributed amongst all the principals in the system, it is difficult for the resource owner to determine what that access control policy

is. A simple solution to this is for the owner of each resource to maintain a database of issued capabilities and their expiration dates. Newly issued capabilities are added to the database while expired capabilities are removed from it. The database will give the resource owner a snapshot of the current access control policy for a given resource. This approach assumes that principals that are authorised to further delegate the access rights will somehow register those delegations with the resource owner in order to keep the database up to date. Compared to an ACL, the resource owner must explicitly maintain a database of issued and valid capabilities, whereas in an ACL-based system the ACL of an object provides a view of the currently active access control policy for that object implicitly. Notice that in many cases, especially in large-scale systems like MAIA, an access control policy would be defined explicitly and capabilities and ACLs would simply be the method for implementing that access control policy. But in smaller systems, e.g. filesystems or web sites, the ACL and capabilities actually represent the access control policy.

Second, capabilities are difficult to revoke. In an ACL-based system the resource owner can simply change the centralised ACL in order to revoke a principal's access rights for future requests. Not so in a distributed capability-based system: once a capability is given to the principal it is impossible to take it away from her. In a digital world the principal is able to make an infinite number of identical copies of the capability and it is practically impossible for the resource owner to delete all of them without relying on tamper-proof devices. Solving this problem is analogous to solving the digital rights management problem.

Fortunately various methods exist to control the validity of issued capabilities. The following sections will describe each of these methods in turn.

6.2.1 Validity Period

The simplest and most reliable method for determining the validity of a capability is the validity period assigned to the capability at the time of issue. The validity period specifies the *not before* and *not after* dates that together define a time span during which the capability is considered to be valid. The verifier will treat the capability as invalid both before the *not before* and after the *not after* dates. If either date is not specified, the validity date is unbounded in that direction.

The method is very reliable and fast, because the decision-making does not rely on third parties such as certificate revocation lists (See §6.2.2) or on-line checks (See §6.2.3). All other more sophisticated revocation methods add complexity to the access control architecture [Aur99]. These methods imply frequent network communication towards third party on-line services, constant availability of those services, and signature generation and verification of generated responses. Therefore, in most systems, we would like to avoid other revocation methods as much as possible and rely only on validity dates. The TAOS operating system is an example of an architecture that relies solely on validity dates for capability revocation [WABL94].

The issuer of a capability can decide to implement policy management based only on va-

lidity periods by issuing so called *short-lived* capabilities that are valid only for a relatively short period of time. Once the capability has expired the principal is expected to request a fresh capability from the issuer. Each time the principal requests a new capability to replace an old expired one, the issuer is able to revoke the principal's access to the resource simply by declining to issue a new capability. Obviously the issuer must wait for the existing capability to expire before a policy change takes effect and therefore the issuer is forced to make a trade-off between the effort required to re-issue capabilities when the policy has not changed, and the time it takes for a policy change to take effect after the policy has been modified.

Whether fast revocation is a requirement or not depends on the application. For example, in the Stock Ticker example in §1.2.1 the stock exchange might implement a pre-paid model in which a brokerage firm is required to pay for access to the stock ticker data in advance. The expiration date on the capability issued to the brokerage firm will be set to match the date when the current contract, which the brokerage firm has already paid for, runs out. This allows the stock exchange to review its access control policy whenever the capability expires without any risk of loss of revenue due to a brokerage firm missing its payments.

In the congestion control example Detective Smith's right to access the Numberplate event stream is based on her obtaining a court order to do so. The warrant has an expiration date set by the judge who issued it. If the court order is revoked because of an appeal, or Detective Smith is removed from the case, the access rights granted to Detective Smith should be revoked immediately. Since the capability was issued to Detective Smith with the assumption that she would not be removed from the case and that the court order would not be revoked, the expiration date on the capability will be based on the expiration date of the court order. To be able to handle these extraordinary circumstances the Metropolitan domain must be able to revoke Detective Smith's access rights before they have expired. The following sections will describe the alternatives that are available for revoking a signed capability before it has expired.

6.2.2 Certificate Revocation Lists

An early form of Certificate Revocation Lists (CRLs) is based on black lists used by banks and credit card companies where a book was distributed to all retailers listing all bad checking accounts and credit card numbers [EFL+99]. The retailer was then responsible for checking that the checking account or credit card used by a customer was not listed in the book before accepting it. Similarly a CRL contains all identities of the certificates that have been revoked. Each verifier is provided with a copy of the latest CRL and it is responsible for checking that the certificate used by a principal has not been revoked. The model allows the issuer to release a new CRL on demand whenever a certificate is revoked. The CRLs have sequence numbers that allow the verifiers to tell which CRL is the most recent one. Whenever a new CRL is released it replaces the previous one at the verifier.

The main problem with this form of CRL is the fact that the revocation process is not de-

terministic: depending on how the new CRL is delivered to the verifiers, a revoked certificate might be accepted as valid if the latest CRL has not yet reached this particular verifier. Moreover, an active adversary could prevent the CRL from ever reaching a given verifier simply by tampering with the network regardless of whether the network connection is encrypted or not.

6.2.3 SPKI On-Line Tests

The SPKI working group wanted to define a set of revocation methods that would allow deterministic revocation behaviour. The SPKI Certificate Theory RFC [EFL+99] specifies three types of deterministic on-line tests:

1. Timed CRLs
2. Timed Revalidations
3. One-Time Revalidations

Timed Certificate Revocation Lists

Timed CRLs aim to address the shortcomings of traditional CRLs described above by attaching a validity period to the CRL that specifies the time period for which the CRL should be considered valid. If the CRL expires and the verifier does not have access to a new valid CRL, it will consider all certificates as invalid. The validity periods of two CRLs must not intersect, i.e. the validity period of the new CRL must start after the previous CRL has already expired. Obviously the replacing CRL can be distributed before it is valid, but the earlier the CRL is distributed the more out of date it will be when it does become valid.

SPKI certificates must state the key that is used to sign the CRL and also where the CRL can be fetched. Unlike with traditional CRLs where the certificate issuer distributes the CRL to the verifiers, in SPKI it is the principal's responsibility to fetch the latest CRL and provide it to the verifier with her other credentials. These rules guarantee deterministic behaviour wherein all certificates that rely on CRLs will always be processed with a valid CRL and CRLs are always issued in a deterministic manner.

Timed Revalidations

A *timed revalidation* is a positive version of a timed CRL. Where a CRL states which certificates have been revoked, a timed revalidation specifies which certificates are still valid. Again, to provide deterministic behaviour, timed revalidations must follow the same rules as described above for timed CRLs.

One-Time Revalidations

Both timed CRLs and timed revalidations force the issuer to make a trade-off between how often to issue a new validity statement and the time it takes for a policy change to take effect. In some cases any latency, however small, is unacceptable. To address these scenarios, the SPKI standard defines *one-time revalidations*. A one-time revalidation states that the certificate mentioned in the instrument is valid *now* for the current authorisation computation only. Because the response from the revalidation service has no validity period, the revalidation request (including a unique nonce) must be generated by the verifier at the time of access. With both timed CRLs and timed revalidations the principal is responsible for retrieving the CRL or revalidation and providing it for the verifier with its other credentials.

6.2.4 Active Revocation

Richard Hayton describes in his PhD dissertation [Hay96] a notification based approach to credential invalidation, which he calls *active revocation*. The idea is that credential issuers notify registered verifiers when a given credential's validity state changes. For example, when making an access control decision a verifier registers at the issuer of a particular credential. The issuer will from then on notify the verifier if the credential's validity state changes, i.e. the credential is revoked. The registration phase in the proposed scheme allows verifiers to retrieve up to date validity information from the issuer that is used in the initial access control decision, and the notification phase guarantees that verifiers are notified as soon as the validity state of the credential changes.

Active revocation was developed to be used in conjunction with an OASIS role server. In such a deployment verifiers register at the OASIS server to be notified if the validity state of a given credential changes. A credential is revoked if one of the pre-conditions in any of the OASIS policy rules that have led to the principal acquiring the given credential becomes false or any of rules are removed from the system. The preconditions, as described in §6.1, are role memberships, appointment certificates, and environmental constraints. If any of the principal's appointment certificates are revoked, or an environmental constraint is no longer true, the OASIS rule will no longer be valid. The invalid rule will trigger a cascading invalidation of all the other rules that depend on this rule being valid. As a result those credentials of the principal that depended on the invalidated rule will be revoked. The OASIS server then notifies all registered verifiers that the credential has been revoked.

For example, in the Detective Smith's case a verifier would register with the Met domain to receive notifications if either the role membership rule or the authorisation becomes invalid. The role membership rule depends on the court order, Detective smith being a detective, and her being assigned to the investigation. The role membership rule would become invalid if (i) the court order was revoked, (ii) Detective Smith lost her detective status, or (iii) Detective Smith was removed from the investigation. The authorisation rule depends only on Detective Smith

being a member of the *npTracker* role, so if Detective Smith lost her membership of the role, she would also lose her numberplate tracking rights.

To avoid the non-determinism typically related to CRL implementations, Hayton's approach implements a heartbeat protocol between the server and the registered verifiers. The heartbeat messages act as timed CRLs in that they refresh the status quo for the next time interval.

Because the verifiers communicate directly with the OASIS server, the parties end up being tightly coupled to each other. The client-server nature of the communication also presents a problem in two fronts: (i) the server must send one message for each registered verifier, and (ii) each verifier must connect to each server in the system.

By using a publish/subscribe system as the notification service we are able to decouple the verifiers from the issuers while maintaining the deterministic behaviour of the revocation system. The publish/subscribe system allows the verifiers to access all servers simply by connecting to a single broker. The servers are able to publish a single revocation message that is delivered to all registered verifiers. We will describe our approach in more detail in the next section.

6.3 Distributing Validity Statements over Publish/Subscribe

Active revocation relies on a one-to-many notification service that could be easily and efficiently implemented over a publish/subscribe system. Verifiers could subscribe to a publish/subscribe topic that was used to publish validity statements for a specific credential or all credentials issued by the same principal issuer depending on the desired granularity.

A crucial step in the active revocation scheme is the initial request phase where the verifier registers to be notified of validity changes. That phase allows the issuer to provide the verifier with the credential's current validity state as a response to the registration request. Unfortunately, in decentralised publish/subscribe the subscribers are decoupled from the publishers so the verifier is unable to make the initial request to determine the current validity state of a credential over the publish/subscribe protocol.

This problem can be addressed in one of two ways: (i) the principal can provide the verifier with the initial validity statement, this is the approach proposed in the SPKI RFC [EFL+99], or (ii) the publish/subscribe protocol is enhanced to support a request-response type interaction model that allows the verifier to receive the initial validity statement over the publish/subscribe system as a response to an initial request. The two approaches can also be combined so that the verifier will rely on receiving the initial validity statement over the publish/subscribe system if the principal did not provide one with its other credentials.

The following proposal is aimed at topic-based publish/subscribe systems, which allows us to ignore the contents of the publications. We will try to generalise our approach for content-based and type-based publish/subscribe systems in the future. Please keep in mind that the

proposal has not been fully fleshed out at the time of writing and that we will aim to publish the work independently of this dissertation.

6.3.1 Request-Response over Publish/Subscribe

In order to use the publish/subscribe system for delivering the initial validity statement to the verifier, we can enhance the publish/subscribe protocol by extending the semantics of the subscription message to include an initial state request. That is, a subscription message will create subscription state in the brokers that it passes through as usual, but it will also double as a request for the current state of the topic. The event service will provide a response to the request and deliver it to the subscriber as a response to the subscription request. After that the protocol falls back to the normal publish/subscribe semantics and the subscriber is notified whenever a new event is published.

In the validation scenario revocation events would be published under credential-specific topics, i.e. each credential would have its own publish/subscribe topic. This would make it simple for the verifier to subscribe to revocation events for a given credential. When a verifier subscribes to validity events for a given credential, the subscription doubles as a request for the current validity state of that credential.

The problem in integrating a request-response protocol into the subscription phase of publish/subscribe is deciding who should provide the response for the request. We can route the request back to a publisher and expect the publisher to provide a response. This works very well and is conceptually simple if we have only one publisher for each topic. This is true for our example scenario where the validity request would be handled by the sole publisher of revocation events under this particular topic.

But in other scenarios we might have more than one publisher for a given topic. Even in the revocation example we might have multiple publishers for reliability's sake. With multiple publishers we can route the request to all of the publishers or pick a subset of publishers at random. When routing the request to all of the known publishers we must decide which response to accept. It is plausible that the publishers do not actually agree on the current state of affairs. We can address this by simply attaching a timestamp to the state reply signifying the age of the provided state and then allow the subscriber (or the SHB) to pick from all the responses the one with the most recent timestamp. If the subscriber is expected to pick the response with the most recent state snapshot it implies that the subscriber must wait for responses from all known publishers. The problem here is that the decentralised event service does not actually know how many publishers there are in the system. Therefore the only alternative for the subscriber is to wait for responses until a timeout expires and then pick one response from the received responses. This approach introduces a constant delay to all subscription requests, which, depending on the length of the delay, might not be acceptable. The other simpler alternative is to always pick the first response in which case the subscriber has to wait for the full timeout only

in cases when there are no publishers in the system. The obvious downside is that the accepted response might not be accurate.

Instead of proposing a solution here, we will add response caching to the protocol in the next section. Response caching allows us to avoid having to pick a publisher in cases when a response has already been provided. We need to actually address the problem only when a cached response is not available.

6.3.2 State Caching

Because event publications follow a predefined event dissemination tree that has been created by advertisement and subscription events flowing though the event broker network, we can use the event dissemination tree to cache previous publications in the broker network. Each new subscription event will eventually reach a node that is a part of the event dissemination tree. Assuming that a broker has cached the most recent publication or response to a previous state request, it can provide an up-to-date state response to the subscriber without having to forward the request any further in the broker network.

When a subscription event is routed through the broker network it will always eventually reach the event dissemination tree, at the very latest when it reaches the rendezvous node for that publish/subscribe topic. By caching previous state responses in each event broker that stores subscription routing state for that publish/subscribe topic we can increase the state response performance of the whole publish/subscribe system.

All caching brokers are free to discard items from their cache if they are running out of resources. If a broker does not have the previous state cached for a given topic, it can always propagate the state request to its parent in the event dissemination tree, all the way up to the rendezvous node. If the rendezvous node does not have the current state, we will have to deal with the problem described in the previous section. One approach, in addition to the ones described previously, is to simply let the rendezvous node respond with an *unknown* message.

If we decide to implement one of the approaches from the previous section we should implement it at the rendezvous node rather than at the subscriber. The rendezvous node represents a root node in the event dissemination tree towards the subscribers. If the rendezvous node makes the request towards the publisher or publishers, the response will immediately be cached by the rendezvous node for the benefit of future subscribers. Thereby we can avoid forwarding requests to the publishers as much as possible.

6.3.3 Publishing Validity Statements

State response caching provides the best performance when as many subscribers as possible are interested in the same topic. In the extreme case when there is only one subscriber per topic the state request will degrade to the worst case scenario where state requests are always routed to the rendezvous node and in many cases all the way to the publishers.

In order to take advantage of state caching in the credential revocation scenario described earlier in this section, we would like one publish/subscribe topic to represent as many credentials as possible. Therefore, we would prefer issuers to publish timed CRLs that cover all the credentials that they have issued and that have not yet expired. That is, each publication would be an SPKI-style timed CRL that states which of those certificates issued by the issuer, that have not yet expired, have been revoked. Each timed CRL would also state its own validity period as required by the SPKI specification.

Because each issuer publishes events only under one topic it allows the broker network to cache the current state, i.e. the previously published CRL, and provide it to new subscribers as the current state. The caching scheme could be further enhanced by making the brokers aware of the CRL's validity semantics so that brokers would be able to purge expired events, i.e. expired CRLs, from their caches when they have expired

In addition to enabling more efficient caching, the one-CRL-per-issuer scheme allows the issuer to offload some processing to the verifiers. That is, the issuer can sign one large statement instead of multiple small ones (one per issued certificate), whereas the verifiers have to verify the signature of a larger document covering a number of certificates instead of a small document covering only the certificate the verifier is interested in.

As stated above this is still work in progress and we have not fleshed out all the details yet. We do feel strongly that the proposed mechanism for caching state in the event service as well as providing the last known state as a response to a subscription are valuable mechanisms for any state related publish/subscribe applications.

6.4 Policy Evaluation at the Local Broker

In addition to static policy rules, the principal's access rights to the publish/subscribe system may also depend on dynamic conditions such as the time when an event was published, or the frequency of publications, to name a few. For example, a publisher may be restricted to publish events only during working hours or if they are on duty at that time.

In order to enforce access restrictions outside of working hours in our current model the publisher's authorisation certificate would expire at e.g. 5pm each day. The publisher would then have to acquire a new certificate that would be valid between 9am and 5pm the next day. If the publisher left early that day, i.e. she were not on duty anymore, the issuer would revoke her credential before it expired. This kind of policy would be easy to specify in OASIS with environmental constraints that evaluate to true between 9am and 5pm or if the principal is on duty.

In order to allow longer lived certificates and to lower the load on the certificate issuer, we can move the evaluation of the dynamic part of the policy to the verifier. That is, instead of issuing certificates with validity times between 9am and 5pm, the issuer could issue a certificate

that was valid for the coming month, but only when the access request was made between 9am and 5pm. Similarly, the verifier could check that the publisher is on duty at the time of publication by querying a database.

Delegating the evaluation of the relatively simple dynamic conditions to the broker would require us to define a minimal and safe policy language that allows the issuer to define these conditions in the authority field of authorisation certificates. The verifier could then evaluate the condition in the authority field and grant access if the condition evaluates to true. The policy language must be "safe" in the sense that it does not allow access to any system calls and the programs are executed in a sandbox that guarantees that the programs behave well (i.e. do not run in an infinite loop or consume large amounts of memory). The programs would receive the requested action (e.g. publish or subscribe) and the current environment (e.g. time, event type) as input. Defining such a policy language is part of our planned future work.

6.5 Distributing Capabilities

When capabilities are implemented as digitally signed certificates, the certificates can be stored and managed anywhere in the system as long as they are available to the verifier at the time of the access request. The digital signature on the capability guarantees the integrity and authenticity of the capability. Therefore, both the principal and the verifier can trust the content of the capability.

Typically it is the principal that gathers evidence and provides that evidence to the verifier at the time of access. The evidence includes capabilities and validity instruments (See §6.2.3). This approach makes it the principal's responsibility to locate and obtain valid, i.e. up to date, credentials and validity instruments. The verifier is responsible only for making the access control decision based on the evidence at hand. The exception to this rule is when the system requires one-time revalidations, as described in §6.2.3.

6.5.1 Gathering Evidence

Before a principal can use a capability that has been issued to her to access a resource, she must gather other evidence, as described above, that together with her capability proves her authority to access the resource. The evidence includes other capabilities that link the principal's capability to the resource owner, and possibly validity statements specified in the capabilities' validity fields.

Assuming that the issuer did not provide the principal with the capabilities that link her to the resource owner, the principal must somehow obtain the missing capabilities that are part of the capability chain. Alternatives include requesting the capabilities from the issuer, or retrieving them from some sort of capability repository. In most cases, including our implementation in

MAIA, we would expect the issuer to provide the principal with the newly issued capability and the chain of capabilities linking that capability to the resource owner.

In order to obtain the required validity statements, the principal needs to look at the validity fields of all the capabilities in the capability chain. The validity field in the capability will specify a URL from where the principal can obtain the required validity statements.

A capability can also define an on-line check that will allow the principal to obtain a fresh capability when the original capability has expired [EFL+99]. This mechanism can be used to refresh any capabilities in the capability chain.

6.5.2 Distribution Methods

Since it is the principal's responsibility to obtain valid credentials before making an access request, it makes sense in most cases to employ pull-type communications between the principal and the credential sources.

In some cases, though, the issuer has a vested interest in the principals having valid capabilities. For example, it is in the domain's best interest to make sure that all brokers and sub-domains have valid credentials. Also, when using group subjects to delegate access rights to event brokers, the domain wants to push new credentials towards all the event brokers at the same time. In these scenarios the issuer should employ push-type messaging between itself and the principals to ensure that the principals have the latest credentials. It is also advantageous from a performance point of view to broadcast the credentials to all principals at the same time rather than wait for each principal to make a request for the same credential.

We assume that in general capabilities and validity instruments will be transmitted out-of-band in the system, i.e. they will be sent to their destinations outside of the publish/subscribe system being protected. It would probably be possible use the publish/subscribe system in implementing the push-type messaging described above to push capabilities to the event brokers in a domain, but we have not yet investigated doing so. Another efficient alternative inside a single domain would be to use multicast to push capabilities to event brokers.

6.6 Related Work

We touched upon moving part of the dynamic policy evaluation to the verifier from the ACS in §6.4. We are proposing that some of the dynamic environmental constraints should be incorporated in the issued credentials and evaluated by the verifier at the time of access. In order to do that we would have to develop a safe policy language that could be embedded in SPKI authorisation certificates. Both the PolicyMaker [BFL96] and KeyNote [BFK98, BFIK99a] systems also rely on such safe policy languages. We would also have to investigate what type of environmental constraints can be implemented reliably at the verifier. For example, the OASIS

environmental constraints can be anything from database lookups to the current time. Many of these environmental constraints could be checked at the verifier when the request is made.

In §6.3 we sketch out mechanisms for (i) using the subscribe event as a request for the current state, or the most recent publication if you will, and (ii) caching the current state, or the most recent publication, in the decentralised event service in order to improve performance. Other work that provides similar functionality in publish/subscribe systems include so called *event replay* services where the event service remembers the n last publications and *replays* those publications to a new subscriber. Many systems that support disconnected operation, i.e. the subscriber is able to disconnect the system while still maintaining its subscriptions, support event replay for those events that were published while the subscriber was disconnected [ZF03] (we discussed disconnected operation in §2.1.1). The state caching approach is suitable for applications that monitor the current state of an object for example, whereas event replay is more suitable for applications that collate a sequence of events.

6.7 Summary

In this chapter we have addressed many issues that we mentioned only briefly in previous chapters. Many of those issues are directly or indirectly related to how access control policy is managed in the system. We provided a cursory look at how the OASIS RBAC system could be integrated with MAIA in order to manage access control policy in domains.

We also addressed credential revocation both from the point of view of SPKI authorisation certificates as well as from an OASIS view point. The revocation of capabilities has traditionally presented trade-offs between certificate lifespans, the load caused by on-line checks, and the timeliness of credential revocation. In some cases the trade-off is non-existent, for example, if strict revocation is not necessary and the application lends itself to short-lived capabilities, then it is simple to issue short-lived capabilities and forgo any other revocation mechanisms. On the other hand applications that require fast revocation of credentials must make frequent validity requests towards the credential issuers thereby creating load on the on-line check service. We feel confident that we have shown that the MAIA architecture enables both the resource owner and the domain to select a policy management mechanism that suits the application's needs.

We also proposed moving some of the dynamic parts of the access control policy evaluation to the verifier by embedding conditions into the SPKI authorisation certificate. This would allow the verifier to evaluate the conditions *in situ* without having to rely on third party services. This would relieve the need for revoking and re-issuing capabilities based on some of the more dynamic conditions.

CHAPTER 7

Event Content Encryption

In Chapter 5 we proposed a capability-based access control architecture for multi-domain publish/subscribe systems. The architecture provides a mechanism for authorising clients to, for example, publish and subscribe to event types. The client's privileges are checked by the local broker hosting the client.

The approach implements access control at the edge of the broker network and assumes that all brokers can be trusted to enforce the access control policies correctly and not to disclose confidential information to unauthorised parties. Any malicious, compromised or unauthorised broker is therefore free to read and write any events that pass through it on their way from the publisher to the subscribers. Malicious brokers are also able to inject and remove messages in the broker network, thereby affecting the routing state of the broker network or creating spurious publications.

This level of access control might be acceptable in a relatively small system deployed inside a single organisation, but it is not appropriate in a multi-domain environment where organisations share a common infrastructure, but do not necessarily trust each other.

In order to enforce access control inside the broker network we propose encrypting event content and controlling access to the encryption keys. With encrypted event content only those brokers that are authorised to access the keys are able to access the event content either to write (i.e. when publishing events) or to read (i.e. when routing or delivering events). We effectively add an extra layer of access control where the key managers control access to the keys.

In addition to protecting the confidentiality of events in unauthorised domains, we can also use encryption to implement a more expressive access control mechanism and lower the number of events sent. By encrypting individual attributes, instead of the whole event as a single block, we are able to enforce attribute level access control in a multi-domain environment: publishers

and subscribers can be authorised to access only a subset of the attributes in an event type. With attribute level access control a single event instance can be delivered to a set of subscribers each with its individual set of access rights. In Chapter 5 we discussed the congestion control example and how the various subscribers require different access rights to the published events. Attribute level encryption will allow us to enforce that level of access control also in the broker network, not just at its edges. Without attribute level encryption the client's domain and the client hosting broker will have access to all the attributes in the event and it is up to the event broker not to disclose the restricted attributes to the event client. Previously we relied on the client hosting broker to enforce attribute level access control for connected event clients. With attribute level encryption the event type owner can enforce the same level of access control towards a domain by not disclosing the appropriate attribute encryption keys.

As stated in §3.2.3, it is assumed that all clients have access to a broker that they can trust and that the broker is authorised to access the event content required by the client. This allows us to implement the event content encryption transparently within the broker network without involving the clients: an event client's local broker encrypts and decrypts events for the event client without the event client ever knowing about it. By delegating the encryption tasks to the brokers, we lower the number of nodes required to have access to a given encryption key. The benefits are four-fold:

i. fewer nodes handle the confidential encryption keys so there is a smaller chance of a key being accidentally or maliciously disclosed

ii. key refreshes involve fewer nodes, which means that the key management algorithm will incur smaller communication and processing overheads to the publish/subscribe system and refreshing encryption keys will be faster

iii. the local broker performs all encryption related tasks for all those event clients that it hosts, thus making the encryption of events transparent from the event clients' point of view

iv. since the local broker decrypts the events, it can decrypt an event once and deliver it to all subscribers, instead of each subscriber having to decrypt the same event

Delegating encryption tasks to the local broker is appropriate, because in this case event content encryption is a middleware feature used to enforce access control within the publish/subscribe system. If applications need to handle encrypted data in the application layer, they are free to publish encrypted data over the publish/subscribe system, i.e. set attributes to already encrypted values.

The rest of this chapter is organised as follows. We present our scheme for event level encryption in §7.1, followed by our scheme for attribute level encryption in §7.2. We propose in §7.3 to encrypt subscription filters in order to avoid leaking information about the events that match a given subscription. In order to minimise the performance impact of encrypting

and decrypting event content, we propose in §7.4 not to encrypt events when they are being forwarded to another broker with equivalent access rights. In §7.5 we discuss details of our implementation of event content encryption in MAIA. Key management is an inevitable part of any system relying on encryption. We discuss using secure group communication based key management in §7.6 and propose a scheme for minimising key refreshes. We evaluate our approach in §7.7. Finally we finish the chapter with a discussion of related work on event content encryption in publish/subscribe systems and a short summary of the chapter.

7.1 Event Level Encryption

In event level encryption all the event attributes are encrypted as a single block of plaintext. The event type identifier, which was discussed in §4.4.3, is left intact in the event (i.e. it is not encrypted). This facilitates faster and more efficient event routing in the broker network, because authorised brokers do not have to decrypt the identifier at each hop, and unauthorised brokers can route the event down the event dissemination tree instead of having to broadcast it to all their neighbours.

The globally unique event type identifier specifies the encryption key used to encrypt the event content. Each event type in the system will have its own individual key (the various versions of the same event type all share the same key, because access is granted to all versions of an event type). Keys are refreshed, as discussed in §7.6.2.

While in transit the event will consist of a tuple containing the type identifier, a publication timestamp, ciphertext, and a message authentication tag[1]: [*type id*, *timestamp*, *ciphertext*, *authentication tag*].

Event brokers that are authorised to access the event, and thus have access to the encryption key, can decrypt the event and implement content-based routing. Event brokers that do not have access to the key will not be able to decrypt the event content and are therefore forced to route the event based only on its type. That is, they will not be able to look at the different attribute values and route the event based on its content.

Event level encryption results in one encryption at the publisher hosting broker, and one decryption at each filtering intermediate broker and subscriber hosting broker that the event passes through, regardless of the number of attributes in the event type. This results in a significant performance advantage compared to attribute level encryption, as we will discuss in §7.7.

[1]The authentication tag is a message authentication code (MAC) produced by the EAX algorithm as described in §7.5

7.2 Attribute Level Encryption

In attribute level encryption each attribute value in an event is encrypted separately with its own encryption key. The key is identified by the attribute's globally unique name (See §4.2).

Similarly to event level encryption, the event type identifier is left intact to facilitate event routing for unauthorised brokers. The attribute identifiers are also left intact to allow authorised brokers to identify each attribute and to decrypt the attribute values with the correct keys. Brokers can implement content-based routing over those attributes that they are authorised to access.

An attribute level encrypted event in transit consists of the event type identifier, a publication timestamp, and a set of attribute tuples: [type id, timestamp, *attributes*]. Attribute tuples consist of an attribute identifier, ciphertext, and a message authentication tag: [attribute id, ciphertext, authentication tag].

Compared with event level encryption, attribute level encryption usually results in larger processing overheads, because each attribute is encrypted separately. In the encryption process the initialisation of the encryption algorithm, including building the key schedule, takes a significant portion of the total running time of the algorithm. Once the algorithm is initialised, increasing the amount of data to be encrypted increases the running time linearly. This disparity is emphasised in attribute level encryption, where an encryption algorithm must be initialised for each attribute separately with the attribute specific key and an event specific initialisation vector, and the amount of data to be encrypted is relatively small. As a result attribute level encryption incurs larger processing overheads when compared with event encryption, which can be clearly seen from the performance results in §7.7.

Because each encrypted attribute has its own authentication tag, attribute level encryption introduces a larger size overhead as well, when compared to event level encryption where an encrypted event includes only one authentication tag.

The significant advantage of attribute level encryption over event level encryption is the higher level of granularity that enables the type owner to control access to the event type at the attribute level. The event type owner can therefore allow different clients to have different levels of access to the same events. Attribute level encryption also enables content-based routing in cases where an intermediate broker has access to only a subset of the event's attributes, thus allowing more efficient event delivery within the broker network. Therefore the choice between event and attribute level encryption is a trade-off between expressiveness and performance, and depends on the requirements of the distributed application. Notice that both event level and attribute level encryption can be implemented in a system at the same time, thereby allowing the event type owner to choose which one to use with a given event type.

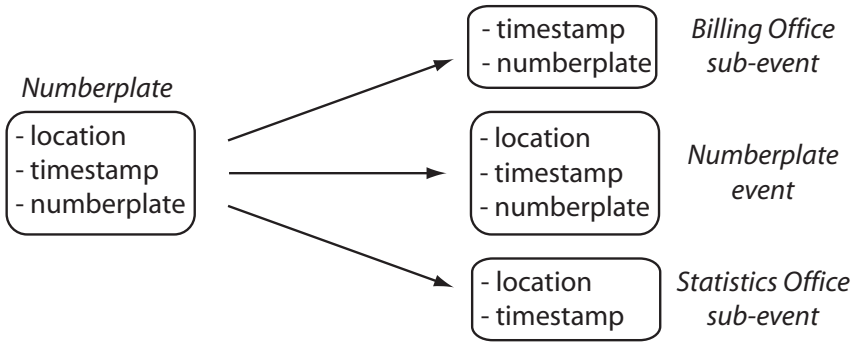

Figure 7.1: In order to emulate attribute level encryption with event level encryption the publisher must publish independent events for all subscriber groups.

7.2.1 Emulating Attribute Level Access Control

One can try to emulate the expressiveness provided by attribute level encryption by introducing a new event type for each group of subscribers that share the same credentials. The publisher would then publish an instance of each of these types instead of publishing just a single event. For example, in the congestion control example, the CCTV cameras would have to publish three events: one for the subscribers with the same authority as the billing office, a second one for the subscribers with the statistics office's authority, and a third one as the full *Numberplate* event for the subscribers with Detective Smith's authority (See Figure 7.1). Each type of sub-event would be encrypted with a separate key that was known to all the subscribers with the same authority. The PHB could implement this transparently so that the publisher has to publish only one event and the PHB would create the appropriate sub-events.

Obviously this approach does not scale as well as attribute level encryption and a large number of subscribers with differing access rights would result in a comparatively larger number of publications, as is shown in §7.7.

The more important performance related aspect of emulating attribute level access control with sub-events is the fact that each group of subscribers sharing the same authority must have their own unique encryption key. The number of these key groups is 2^n, where n is the number of attributes in the event type. On the other hand the number of key groups with attribute encryption is only n, because access to each attribute is granted independently of the other attributes. For example, an event type with 5 attributes has at most 32 subscriber key groups when emulating attribute level access control with event level encryption, whereas attribute level encryption would require only 5 key groups.

7.2.2 Restricted Attribute Values

We described in §5.2.2 how to restrict attribute values for publications and subscriptions. Assuming that the access control policy is enforced in the broker network by encrypting attributes

it is not possible to force domain level restrictions on individual attributes. For example, PITO can not force one domain to publish *Numberplate* events with the location set to `Victoria` in all publications, because an intermediate broker receiving the publication from another broker will not know which broker published the event and therefore cannot verify that the publication meets the attribute value restrictions placed on the PHB.

Restricting attributes on a domain level would mean that the PHB would have to attach its credentials to the publication and sign it. Each broker on the publication's path from the PHB to all SHBs would have to verify the signature, verify the PHBs credentials, decrypt all the event attributes and check that the attribute values conform to the PHB's credentials. This procedure would incur a lot of processing overhead and it would impact the throughput of the broker network quite significantly. Therefore, we assume that restrictions on attribute values are used only when issuing authorisation certificates to event clients where the client hosting broker, who is a member of the same domain as the client, is responsible for enforcing the restrictions.

7.3 Encrypting Subscription Filters

In order to prevent the event content from leaking to unauthorised parties we must also encrypt the filter expressions attached to subscriptions. Encrypted subscription filters guarantee: (i) that only authorised brokers are able to submit subscriptions to the broker network, and (ii) that unauthorised brokers do not gain information about event content, by monitoring which subscriptions a given event matches. For example, in the first case an unauthorised broker can create subscriptions with appropriately chosen filters, route them towards the root of the event dissemination tree, and monitor which events were delivered to it as matching the subscription. The fact that the event matched the subscription would leak information to the broker about the event content, even if the event was still encrypted. In the second case, even if an unauthorised broker was unable to create subscriptions itself, it could still look at subscriptions that were routed through it, take note of the filters on those subscriptions, and monitor which events are delivered to it by upstream brokers as matching the subscription filters. This would again reveal information about the event content to the unauthorised broker.

In the case of event level encryption, we encrypt the complete subscription filter. The event type identifier in the subscription must be left intact to allow brokers to route events based on their type when they are not authorised to access the filter. In such cases the unauthorised broker is required to assume that events of such a type match all filter expressions, i.e. the brokers implement only topic-based routing.

With attribute level encryption each attribute filter is encrypted individually, similarly to when encrypting publications. In addition to the event type identifier the attribute identifiers are also left intact to allow authorised brokers to decrypt those filters that they have access to, and route the event based on it matching the decrypted filters.

7.3.1 Coverage Relations with Encrypted Filters

In order to take advantage of subscription coverage in type and content-based publish/subscribe systems when encrypting attributes, we extend the coverage relation to handle publication and subscriptions with encrypted attributes and filter expressions.

We treat the filter expression in a subscription as a conjunction of attribute filters, as described in §2.2.2. When we employ attribute encryption, each attribute and attribute filter is encrypted with an attribute-specific key. An encrypted attribute is covered by an encrypted attribute filter if the filter matches the attribute value and the event broker applying the filter has access to the encryption key for that attribute. Thereby an attribute encrypted publication is covered by an encrypted subscription when the event broker has access to the encryption keys of all the attribute filters and each of those filters covers the attribute value in the publication:

$$p \sqsubset_S^P f \Leftrightarrow \forall \phi_k \in f : \exists \alpha_k \in p : \exists k \in K : \alpha_k \sqsubset_f^p \phi_k,$$

where K is the set of encryption keys that are available to the event broker, and ϕ_k and α_k represent filters and attributes encrypted with the key k, respectively. Notice that the event broker applying the subscription to the publication does not have to have access to all the attributes in the publication. It is enough for the event broker to have access to all the attribute keys that are included in the subscription filter.

The subscription coverage relation is defined in terms of the publication coverage relation. Therefore the subscription coverage relation stays the same even in the presence of encrypted filters and attributes:

$$f_2 \sqsubset_S^S f_1 \Leftrightarrow \forall p \in P : p \sqsubset_S^P f_2 \Rightarrow p \sqsubset_S^P f_1,$$

7.4 Avoiding Unnecessary Encryptions and Decryptions

Encrypting the event content is not always necessary. If the current broker and the next broker down the event dissemination tree have the same credentials with respect to the event type at hand, we can pass the event to the next broker in plaintext. For example, as argued in §5.1.3, one can assume that in most cases all brokers inside an organisation would share the same credentials. Therefore, as long as the next broker is a member of the same domain, the event can be routed in plaintext. With attribute level encryption it is possible that the neighbouring broker is authorised to access a subset of the decrypted attributes, in which case those attributes that the broker is not authorised to access would be passed to it encrypted.

In order to know when it is safe to pass the event in plaintext form, the brokers exchange

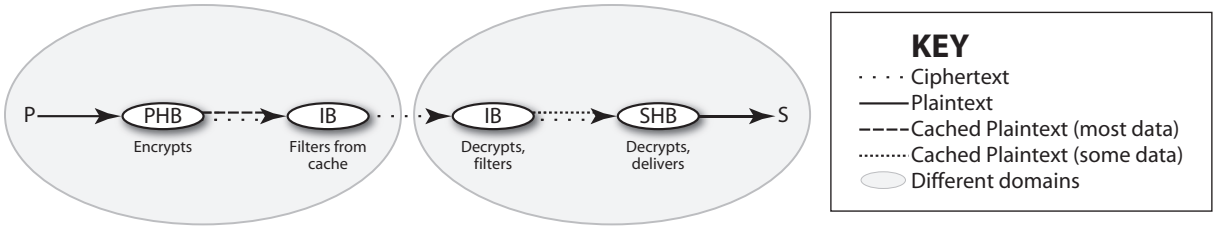

Figure 7.2: Caching decrypted data can increase efficiency when delivering an event to a peer with similar privileges.

credentials as part of a handshake when they connect to each other (See §5.3). When a broker verifies the credentials of one of its neighbouring brokers, it adds those credentials to the routing table entry for that broker for future reference. If a broker acquires new credentials after the initial handshake, it will present these new credentials to its existing neighbours as soon as possible by performing part of the handshaking protocol again.

To avoid unnecessary decryptions, we attach a plaintext content cache to encrypted events. A broker fills the cache with content that it has decrypted, for example, in order to filter on that content. The cache is accessed by the broker when it delivers an event to a local subscriber after first matching the event against the subscription filter, but the broker also sends the cache to the next broker together with the encrypted event. The next broker can look up the attribute from the cache instead of having to decrypt it. If the event is being sent to an unauthorised broker, the cache will be discarded before the event is sent. Obviously sending the cache with the encrypted event will add to the communication cost, but this is outweighed by the savings in encryption/decryption processing. In Figure 7.2 we see two separate cached plaintext streams accompanying an event depending on the inter-broker relationships in two different domains.

Regardless of its neighbouring brokers, the PHB will always encrypt the event content, because it is cheaper to encrypt the event once at the root of the event dissemination tree. In Hermes the rendezvous node for each event type is selected uniformly randomly (the event type name is hashed with the SHA-1 hash algorithm to produce the event type identifier, the identifier is used to select the rendezvous node in the structured overlay network, as described in §2.3.1). Because SHA-1 values are indistinguishable from random values, it is probable that the rendezvous node will reside outside of the current domain. This situation is illustrated in the event dissemination tree in Figure 7.3. Therefore, even with domain internal applications where the event can be routed from the publisher to all subscribers in plaintext form, the event content will in most cases have to be encrypted for it to be routed to the rendezvous node.

We show in §7.7 that the overhead of sending an encrypted event with a full plaintext cache incurs almost no processing overhead compared to sending plaintext events.

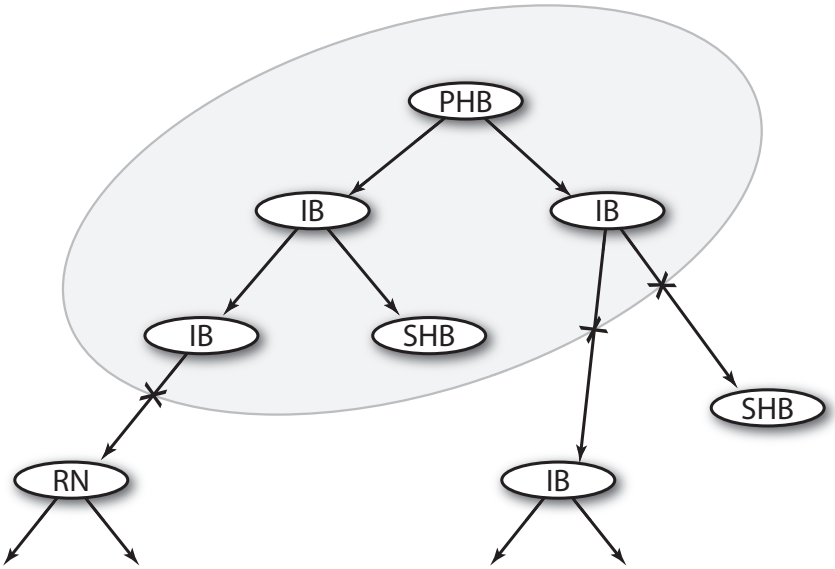

Figure 7.3: Node addressing is effectively random, therefore the rendezvous node for a domain internal type can be outside of the domain that owns an event type.

7.5 Implementation

In our implementation we used the EAX mode [BRW03] of operation when encrypting events, attributes, and subscription filters. EAX is a mode of operation for block ciphers, also described as an *authenticated encryption with associated data* (AEAD) algorithm, that provides simultaneously both data confidentiality and integrity protection. The algorithm implements a two-pass scheme where during the first pass the plain text is encrypted, and on the second pass a *message authentication code* (MAC) is generated for the encrypted data.

The EAX mode is compatible with any block cipher. We used the *advanced encryption standard* (AES) [FIP01] algorithm in our implementation, because of its standard status and the fact that the algorithm has gone through thorough cryptanalysis during its existence and no serious vulnerabilities have been found thus far.

In addition to providing both confidentiality and integrity protection, the EAX mode uses the underlying block cipher in *counter mode* (CTR mode) [DH79, LRW00]. A block cipher in CTR mode is used to produce a stream of key bits that are then XORed with the plaintext. In effect the CTR mode of operation transforms a block cipher into a stream cipher. In our application the advantage of stream ciphers when compared to block ciphers is that the ciphertext is the same length as the plaintext, whereas with block ciphers the plaintext must be padded to a multiple of the block cipher's block length (e.g. the AES block size is 128 bits). Avoiding padding is very important in attribute level encryption, because we encrypt single attributes that might be very small in size. For example, a single integer might be 32 bits in length, which would be padded to 128 bits if we used a block cipher. With event level encryption the message expansion is not that relevant, since the length of padding required to reach the next 16-byte multiple will

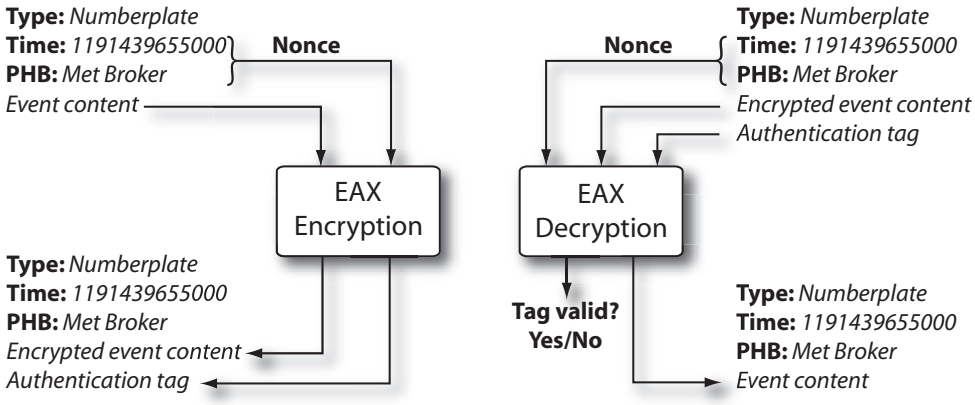

Figure 7.4: The EAX mode of operation.

probably be relatively small compared to the overall plaintext length.

In encryption mode the EAX algorithm takes as input a nonce, a key and the plaintext, and it returns the ciphertext and an authentication tag. In decryption mode the algorithm takes as input the key, the ciphertext and the authentication tag, and it returns either the plaintext, or an error if the authentication check failed. The EAX inputs and outputs can be seen in Figure 7.4.

The nonce can be of arbitrary length. It is expanded (or compressed) to the block length of the underlying block cipher by passing it through an OMAC construct (One-key MAC [IK03]). The OMAC takes an arbitrary length input and produces a fixed length output. In EAX the output of the OMAC construct is used as the initialisation vector for the CTR mode of operation. The OMAC construct guarantees that small changes in the nonce result in large changes in the initialisation vector.

It is important that particular nonce values are not reused, otherwise the block cipher in CTR mode would produce an identical key stream. In our implementation we create a nonce by concatenating the PHB defined event timestamp (64-bit value counting the milliseconds since January 1, 1970 UTC) and the PHB's identity. The timestamp alone is not enough to guarantee unique nonces as two PHBs can publish an instance of the same event type at the same moment in time, thus resulting in two identical key streams. The PHB's identity could be replaced with a PHB-specific random value. The PHB is responsible for making sure that the timestamp grows monotonically from publication to publication. Both the timestamp and the PHB's identity must be included in the published event in order to allow other broker's to decrypt and verify the authenticity of the ciphertext.

The authentication tag is appended to the produced ciphertext to create a two-tuple. With event level encryption a single tag is created for the encrypted event. With attribute level encryption each attribute is encrypted and authenticated separately, and they all have their individual tags. The tag length is configurable in EAX without restrictions, which allows the user to make a trade-off between the authenticity guarantees provided by EAX and the added communication overhead. We used a tag length of 16 bytes in our implementation, but one could make the tag

length a publisher/subscriber defined parameter for each publication/subscription or include it in the event type definition to make it a type specific parameter.

EAX also supports *associated data* that is included in the tag calculation, but is not encrypted. That is, the integrity of the data is protected by the authentication tag, but it is still readable by all principals in the system. In event level encryption the event type should be added to the tag calculation as associated data. Similarly in attribute level encryption, the event type and each attribute name should be included in the tag calculation as associated data for each attribute. We have not had time to implemented this in MAIA yet.

Other AEAD algorithms include the *counter with CBC-MAC mode* (CCM) [WHF03] and the *offset codebook mode* (OCB) [RBBK01]. The CCM mode is the predecessor of the EAX mode. EAX was proposed in order to address some problems that were discovered in the CCM mode [RW03]. Similarly to EAX, CCM is also a two-pass mode. The OCB mode requires only one pass over the plaintext, which makes it roughly twice as fast as EAX and CCM. Unfortunately the OCB mode has a patent application pending in the USA, which restricts its use.

7.6 Key Management

In both encryption approaches the encrypted event content has a globally unique identifier (i.e. the event type or the attribute name). That identifier is used to identify the encryption key to use for encrypting and decrypting the event content. Each event type, in event level encryption, and attribute, in attribute level encryption, has its own individual encryption key. By controlling access to the key we effectively control access to the encrypted event content.

In order to control access to the keys we form a *key group* of event brokers for each individual key. The key group is used to refresh the key when necessary and deliver the new key to all current members of the key group. The key group manager is responsible for verifying that a new member requesting to join the key group is authorised to do so. Therefore the key group manager must be trusted by the type owner to enforce access control correctly. We expect that the key group manager is either the type owner, the ACS of the type owner's domain, some other member of the type owner's domain trusted to manage keys, or a trusted third party that is managing keys for a number of domains in the shared publish/subscribe system.

We proposed a capability-based access control architecture in Chapter 5 where we use capabilities to decentralise access control policy enforcement amongst the publish/subscribe nodes (i.e. clients and brokers): each node holds a set of capabilities that define its authority. The authority to publish or to subscribe to a given event type is granted by the type owner issuing the principal a capability where the capability defines the event type, the action, and the attributes that the node is authorised to access. For example, a tuple [`Numberplate`, subscribe, *] would authorise the owner to subscribe to *Numberplate* events with access to all attributes in

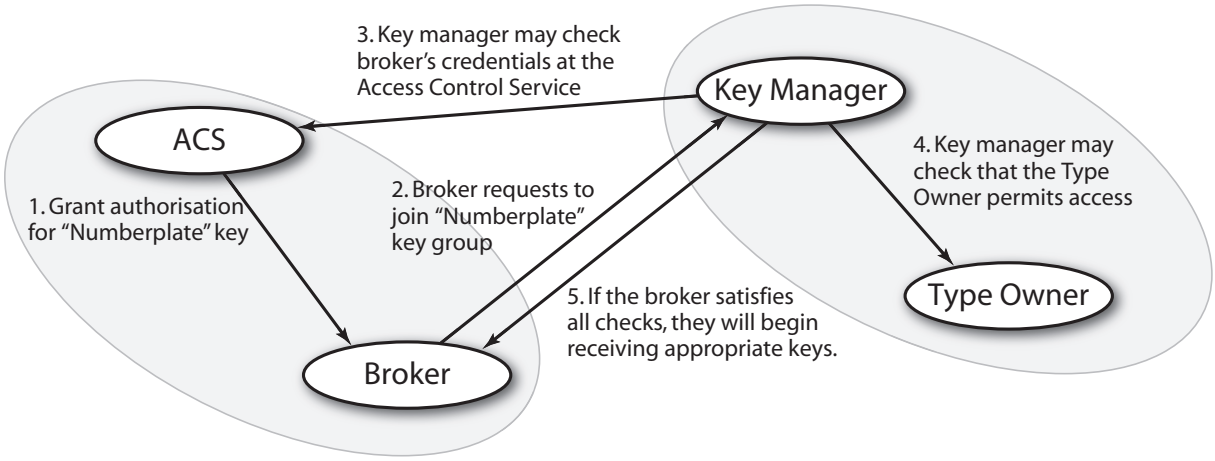

Figure 7.5: The steps involved for a broker to be successful in joining a key group.

the published events.

We use the same capabilities to authorise membership in a key group that are used to authorise publish/subscribe requests (See §5.2.2). Not doing so could lead to inconsistencies where an SHB is authorised to make a subscription on behalf of its clients, but is not able to decrypt incoming event content for them. In the Congestion Control example, the broker hosting a CCTV camera is authorised to join the *Numberplate* key group as well as the key groups for all the attributes in the *Numberplate* event type.

Figure 7.5 shows what steps are required to grant a broker the authority to join a key group and how this authority is verified.

7.6.1 Secure Group Communication

Event content encryption in a decentralised publish/subscribe system can be seen as an instance of secure group communication. In both cases the key management system must scale well with the number of clients, the clients might be spread over large geographic areas, there might be high rates of churn in group membership, and all members must be synchronised with each other in time in order to use the same encryption key at the same time.

There are a number of scalable key management protocols for secure group communication [RH03]. We have implemented the *One-Way Function Tree* (OFT) [SM03] protocol as a proof of concept. We chose OFT for our proof of concept, because it was easy to implement and the performance was only slightly worse than the performance of the more advanced protocols evaluated in [RH03].

Our implementation uses the same structured overlay network used by the MAIA broker network as a transport. The OFT protocol is based on a binary tree where the participants are at the leaves of the tree. It scales in $\log n$ in processing and communication costs, as well as in the size of the state stored at each participant. We have verified this in our simulations.

7.6.2 Key Refreshing

Traditionally in group key management schemes the key is refreshed when a new member joins the group, an existing member leaves the group, or the key refreshing timer expires. Refreshing the key when a new member joins provides backward secrecy, i.e. the new member is prevented from accessing old messages. Similarly refreshing the key when an existing member leaves provides forward secrecy, i.e. the old member is prevented from accessing future messages. Timer triggered refreshes are issued periodically in order to limit the amount of traffic encrypted with the same key. This is important both in limiting the amount of ciphertext available for the adversary to use in cryptanalysis, and the amount of traffic that will be compromised if the current session key were to be compromised.

Even though the state-of-the-art key management protocols are efficient, refreshing the key introduces extra traffic and processing amongst the key group members, often unnecessarily. In our case key group membership is based on the broker holding a capability that authorises it to access a given event type or event attribute and therefore to join the appropriate key group. The capability has a set of validity conditions that in their simplest form define a time period when the capability is valid, and in more complex cases involve on-line checks back to the issuer of the capability.

In some cases a joining broker might have been authorised to access the key at the time of the previous key refresh. In such a case there is no need to force a key refresh when the broker joins the key group, because it is authorised to access the events between the previous key refresh and now. Similarly a leaving broker's credential might be valid for some time after the broker has left the key group. Again, there is no need to refresh the group key until the leaving brokers authority has expired. The key manager can therefore avoid unnecessary key refreshes by looking at the validity conditions of the group member's credentials. With joining brokers the key refresh can be avoided if the broker was authorised to join the key group at the time of the previous key refresh. Similarly with leaving brokers, the key refresh can be deferred until the leaving broker's credentials expire or it is time for the periodic key refresh.

These situations are both illustrated in Figure 7.6. It can be assumed that the credentials granted to brokers are relatively static, i.e. as described in §5.1.3. Therefore, once a domain is authorised to access an event type, the authority will be delegated to all brokers of that domain, and they will have the authority for the foreseeable future. More fine grained and dynamic access control would be implemented at the edge of the broker network between the clients and the client hosting brokers.

When a key is refreshed the new key is tagged with a timestamp. The key to use for a given event is selected based on the event's publication timestamp. The old keys will be kept for a reasonable amount of time in order to allow for some clock drift. Setting this value is part of the key management protocol, although exactly how long this time should be will depend on the nature of the application and possibly the size of the network. It can be configured

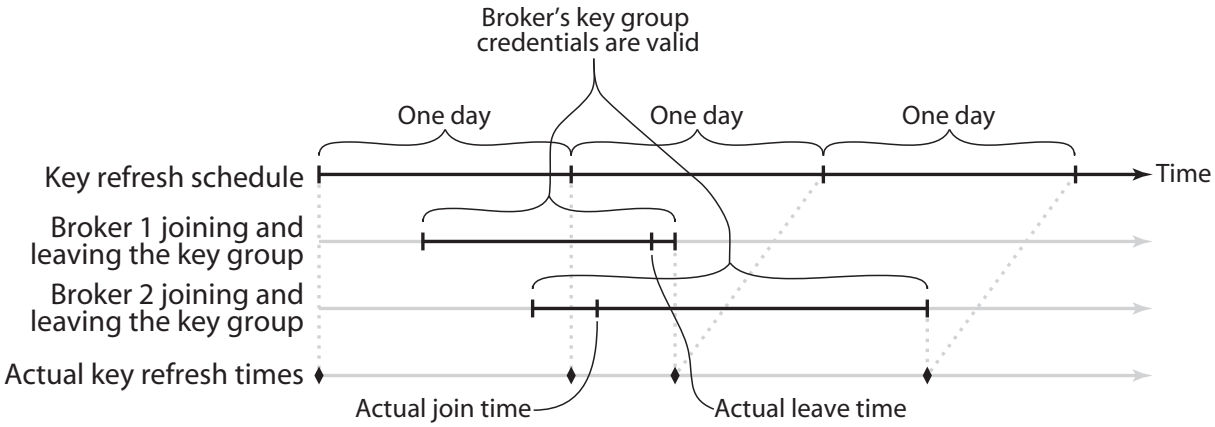

Figure 7.6: Key refreshes can be delayed based on the validity times of the broker's authority.

independently per key group if necessary.

7.7 Evaluation

In order to evaluate the performance of event content encryption we have implemented both encryption approaches in MAIA. The implementation supports all three modes of operation: plaintext content, event level encryption, and attribute level encryption.

We have evaluated the overhead added by the two encryption approaches compared to plaintext events with a number of micro benchmarks. We have not evaluated the routing performance of the system, because it is directly based on Hermes.

We ran three performance tests in a discrete event simulator that is part of the FreePastry distribution [Fre07]. We used a discrete event simulator instead of running the tests over a live system in order to be able to measure the increase in overall processing required to publish a given number of events. The Pastry simulator was run on an Intel P4 3.2GHz workstation with 1GB of main memory. The sections below will describe each specific test in more detail.

7.7.1 End-to-End Overhead

The end-to-end overhead test shows how much the overall message throughput of the simulator was affected by event content encryption. We formed a broker network with two brokers, attached a publisher to one of them and a subscriber to the other one, as shown in Figure 7.7. The subscriber subscribed to the advertised event type without any filters, i.e. each publication matched the subscriber's subscription and thus was delivered to the subscriber. The test measures the combined time it takes to publish and deliver 100,000 events. If the content is encrypted the measured time includes both encrypting the content at the PHB and decrypting it at the SHB.

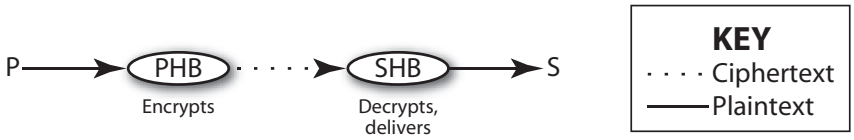

Figure 7.7: The end-to-end test setup.

In the test the number of attributes in the event type is increased from 1 to 25 (the x-axis). Each attribute is set to a 30 character string. For each number of attributes in the event type the publisher publishes 100,000 events, and the elapsed time is measured to derive the message throughput. The test was repeated five times for each number of attributes and we use the average of all iterations in the graph, but the results were very consistent so the standard deviation is not shown. The same tests were run with plaintext events, event level encryption, and attribute level encryption.

As can be seen in Figure 7.8, event content encryption introduces a large computational overhead compared to plaintext events. The throughput when using attribute level encryption with an event type with one attribute is 46% of the throughput achieved when events are sent in plaintext. When the number of attributes increases the performance gap increases as well: with ten attributes the performance with attribute level encryption has decreased to 11.7% of plaintext performance.

Event level encryption fares better, because there are fewer encryption operations per event even though the amount of data to encrypt is larger. The number of individual encryption operations affects the performance more than the amount of data that needs to be encrypted. The difference in performance with event level encryption and attribute level encryption with only one attribute is caused by the Java object serialisation mechanism: in the event level encryption case the whole attribute structure is serialised, which results in more objects than serialising a single attribute value. A more efficient implementation would provide its own marshalling mechanism.

Note that the EAX implementation we use runs the nonce (i.e. initialisation vector) through an OMAC construct. Since the nonce is not required to be kept secret, there is a potential time/space trade-off we have not yet investigated in attaching the output of the OMAC construct to the event. That way the brokers decrypting the event content do not have to run the nonce through the OMAC construct themselves. This optimisation would allow a trade-off to be made between the size of the events and the time it takes to decrypt them. The performance increase might be considerable especially with attribute level encryption where the OMAC construct is executed for each attribute separately even though the nonce is the same in each case.

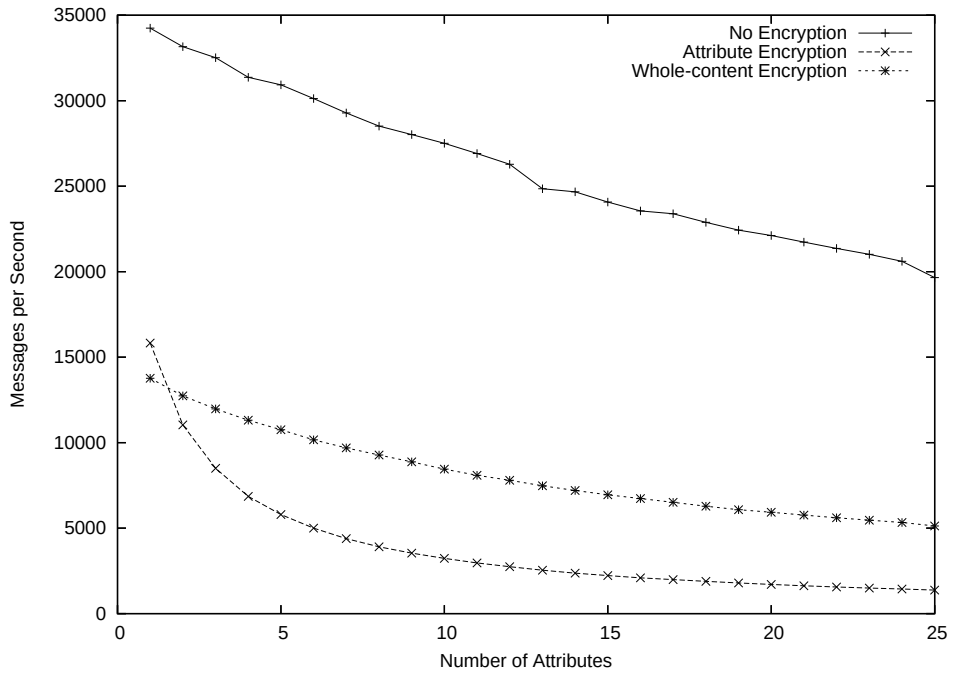

Figure 7.8: The end-to-end throughput of events with plaintext events, event level encryption, and attribute level encryption.

7.7.2 Domain Internal Events

We explained in §7.4 that event content decryption and encryption could be avoided if both brokers are authorised to access the event content. This test was designed to show that by attaching the cached plaintext to the encrypted event when sending an event from one authorised broker to another results in only a small performance overhead when compared to plaintext events.

In this test we again form a broker network with two brokers, as shown in Figure 7.7. Both brokers are configured with the same credentials. The publisher is attached to one of the brokers and the subscriber to the other, and again the subscriber does not specify any filters in its subscription.

The publisher publishes 100,000 events and the test measures the elapsed time in order to derive the message throughput for the system. The event content is encrypted outside the timing measurement, i.e. the encryption cost is not included in the measurements. The goal is to model an environment where a broker has received a message from another authorised broker, and it routes the event to a third authorised broker. In this scenario the intermediate broker is not required to encrypt or decrypt any of the event content, because the PHB provided it with both the plaintext and ciphertext content.

As shown in Figure 7.9, the elapsed time was measured as the number of attributes in the published event was increased from 1 to 25. The attribute values in each case are 30 character strings. Each test is repeated five times, and we use the average of all iterations in the graph. The

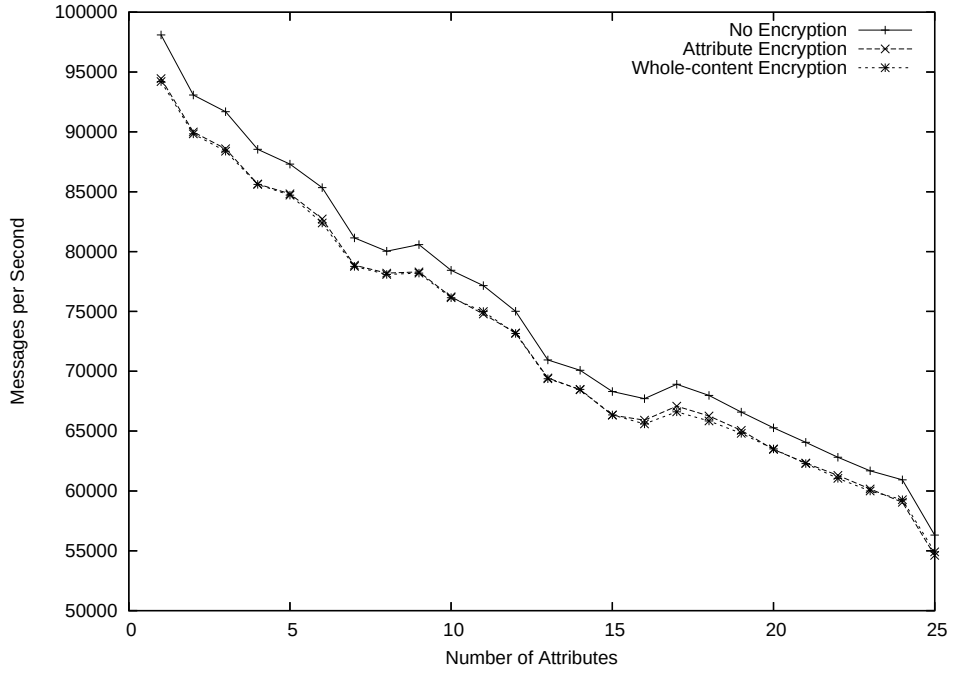

Figure 7.9: The end-to-end throughput of events with plaintext events, event level encryption, and attribute level encryption when plaintext caching is enabled.

same test was repeated with no encryption, event level encryption and attribute level encryption turned on.

The two encrypted modes follow each other very closely. The plaintext mode performs a little better for all attribute counts. The difference can be explained partially by the encrypted events being larger in size, because they include both the plaintext and the encrypted content in this test. The difference in performance is 3.7% with one attribute and 2.5% with 25 attributes.

7.7.3 Communication Overhead

As we explained in §7.2.1, it is possible to emulate the expressiveness of attribute level encryption by defining multiple event types and applying event level encryption of those events. The third test we ran was to show the communication overhead caused by this emulation technique, compared to using real attribute level encryption.

In the test we form a broker network of 2000 brokers. We attach one publisher to one of the brokers, and an increasing number of subscribers to the remaining brokers. Each subscriber simulates a group of subscribers that all have the same access rights to the published event. Each subscriber simulating a group of subscribers has its own event type in the test.

The outcome of this test is shown in Figure 7.10. The number of subscribers is increased from 1 to 50 (the x-axis). For the n subscribers the publisher publishes one event to represent the use of attribute level encryption and n events representing the events for each subscriber group. We count the number of hops each publication makes through the broker network (y-axis). Note

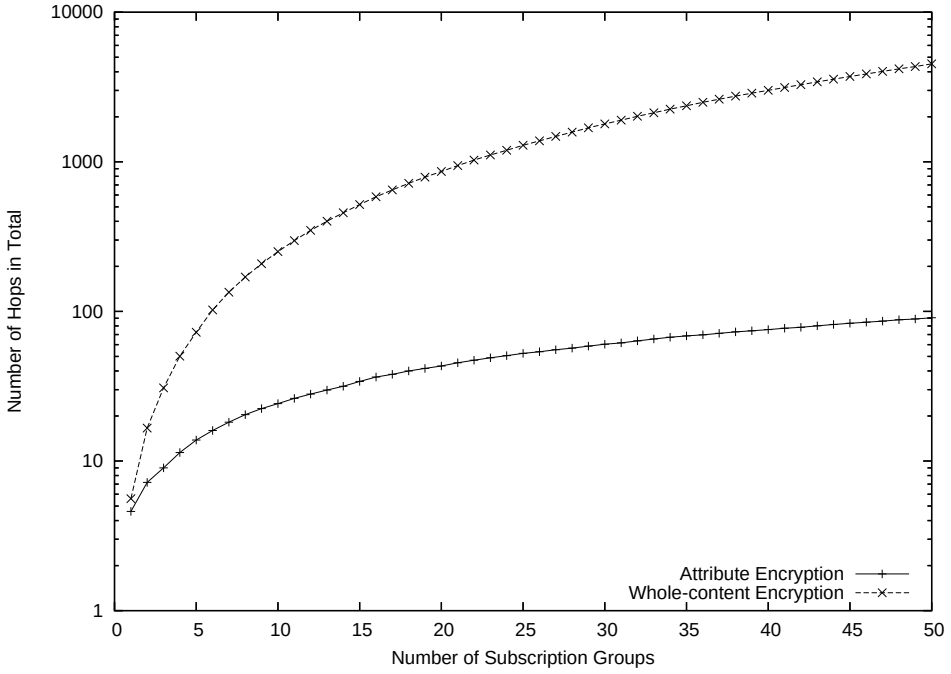

Figure 7.10: The average number of hop counts when emulating attribute level encryption with event level encryption and multiple sub-types ($\log$ scale).

that y-axis is in logarithmic scale.

Note that Figure 7.10 shows workloads beyond those we would expect in common usage, in which many event types are likely to contain fewer than ten attributes. The subscriber groups used in this test represent disjoint permission sets over such event attributes. The number of these sets can be determined from the deployed access control policy. The upper limit will be 2^n, where n is the number of attributes in the event type.

The figure indicates that attribute level encryption scales better than event level encryption even for small numbers of subscriber groups. Indeed, with only three subscriber groups (e.g. the case with *Numberplate* events) the hop count increases from 9.0 hops on average for attribute level encryption to 30.8 hops on average for event level encryption. With 10 subscriber groups the corresponding numbers are 24.2 and 251.0 on average.

7.8 Related Work

Opyrchal and Prakash address the problem of event confidentiality at the last link between the subscriber and the SHB in [OP01]. They correctly state that a secure group communication approach is infeasible in an environment like publish/subscribe that has highly dynamic group memberships. As a solution they propose a scheme utilising key caching and subscriber grouping in order to minimise the number of required encryptions when delivering a publication from a SHB to a set of matching subscribers. We assume in our work that the SHB is powerful

enough to manage a TLS-secured connection for each local subscriber.

Raiciu and Rosenblum present a formal security model for protecting the confidentiality of published events in content-based publish/subscribe systems in [RR06]. The presented model allows the broker network to route events based on their encrypted content. The paper discusses only content-based publish/subscribe systems, but the scheme should be equally applicable to type-based publish/subscribe. Compared to our work, Raiciu and Rosenblum assume that none of the brokers in the broker network are trustworthy. Therefore, event encryption and decryption is implemented by the event clients themselves. In order to implement content-based routing with these assumptions the brokers must be able to apply subscription filters to encrypted content, which is enabled by the proposed scheme. We on the other hand assume that some brokers can be authorised to access event content. Specifically local brokers of event clients are trusted to encrypt and decrypt events for the clients. Intermediate brokers that are trusted to decrypt attributes can use those attributes to implement content-based routing and all brokers are able to route events based on the plaintext type-name in the event. The scheme proposed by Raiciu and Rosenblum could be integrated with our approach in order to allow unauthorised event brokers to also route events based on their content. The downside of the scheme is that it places requirements on the filtering language since some filtering operations cannot be implemented on encrypted content.

Srivatsa and Liu present EventGuard in [SL05]. EventGuard provides event confidentiality, integrity and authenticity in decentralised publish/subscribe systems. The prototype is built on top of Siena [CRW01]. The paper concentrates on topic-based publish/subscribe, but the authors state that the approach is equally applicable to content-based publish/subscribe. In the scheme publishers sign events and encrypt them with a publication-specific, random encryption key. The encryption key is then encrypted with a topic-specific key and attached to the event. Event brokers are expected to verify the signature on each routing hop. The subscriber on receipt of the publication verifies the publisher's signature, decrypts the random key, and finally decrypts the message. A trusted *meta service* (MS) is used to certify advertisement and subscription messages, i.e. it controls access to the event service, but the paper does not address the definition of policy. The paper includes micro benchmarks that show that the excessive use of public key encryption has an effect on performance. For example, each signature verification increases the time needed to handle a single event by 1.7ms. In contrast on the same hardware Siena handles a subscription event in less than 50μs. Similarly to Raiciu and Rosenblum, the model assumes that event brokers are not trustworthy and as a result all encryption operations are implemented by the event clients, whereas our architecture provides encryption as a transparent infrastructure feature to event clients.

In another paper Srivatsa and Liu propose an efficient key management scheme for publish/subscribe systems [SL07]. The work utilises *key graphs* [WGL00] where a root key is hashed in order to generate child keys in a key tree. By partitioning the value space of an attribute in content-based publish/subscribe into a tree, the hierarchical key derivation algorithm can be

used to create keys for subsets of the value space in an efficient manner. The scheme assumes that both publishers and subscribers are authorised to access events based on their content, i.e. they request an encryption key based on what the publication content or subscription filter is. We proposed similar functionality in §5.2.2 that would allow domains to limit the authority of both publishers and subscribers to only certain subset of all events published as instances of a given type depending on the values of attributes. The difference in our approach is that we expect the local broker to enforce the restrictions whereas Srivatsa and Liu are able to enforce the restrictions by preventing the clients from reading/writing events that are within their authority. The key management scheme is applicable in the context of MAIA although we feel that the level of granularity already provided by MAIA (i.e. attribute-level access control) should be enough for most scenarios.

Srivatsa and Liu also address the problem of *frequency inference attacks* in the routing network. Given *a priori* knowledge about event frequency distributions, a broker can infer which type of event the publication is even when the event type has been blinded by hashing it. The proposed solution is to implement probabilistic routing in the broker network so that the number of events flowing through any given broker in the system has been skewed enough to prevent frequency based inference. We have not considered traffic monitoring attacks in our work thus far. Another alternative is to introduce fake events into the event stream. In our case, since we trust the PHBs, we can delegate the generation of fake events to the PHBs. Each PHB can introduce enough fake events into the event stream to skew the event distributions as well as cover for periods when there are no real events published.

Khurana proposes a security scheme in [Khu05] that protects the integrity and confidentiality of some event attributes while allowing subscribers to verify the authenticity of the event. Khurana makes the assumption that only some of the attributes in an event in a content-based publish/subscribe system need to be encrypted while the remaining attributes can remain unencrypted. The proposed scheme implements content-based routing only on the unencrypted attributes thereby forcing the application designers to make a trade-off between efficient routing and event confidentiality. Similarly to EventGuard, the architecture relies on a trusted *proxy security and accounting service* (PSAS) to proxy-encrypt events. That is, the PSAS service will decrypt a publication, verify two signatures, re-encrypt the publication for a given subscriber, and finally sign it. The subscriber hosting broker is expected to pass each received publication to the PSAS service for authentication and re-encryption separately for each matching subscriber. Also, all encryption operations are based on public key algorithms that are several orders of magnitude slower than symmetric equivalents [MOV96]. Khurana expects that each subscriber receives only one event per minute at most. We feel that such a throughput assumption is not realistic in multi-domain systems. For example, in the numberplate monitoring application the Billing Office will see tens of events per second when vehicles enter the congestion controlled area in London. In Khurana's model none of the event brokers are trusted and therefore the PSAS service is used as a trusted third party between the publisher and the subscribers. Be-

cause of the high computational cost associated with each event delivered to a subscriber, a large number of PSAS nodes must be deployed in the system. We would argue that the PSAS functionality should be incorporated in a trusted local broker rather than deployed as a separate entity.

7.9 Summary

Event content encryption can be used to enforce access control policy while events are in transit in the broker network of a multi-domain publish/subscribe system. Encryption introduces a small communication overhead in the form of nonces and authentication tags, and a very large computational overhead in the form of encryption operations. But (i) in many cases there may be no alternatives, i.e. event integrity and confidentiality protection is required, and (ii) the performance penalty can be lessened with implementation optimisations, such as passing cached plaintext content alongside encrypted content between brokers with identical security credentials and thereby lessening the computational overhead by increasing the communication overhead.

Attribute level encryption can be implemented in order to enforce fine-grained access control policies. In addition to providing attribute level access control, attribute level encryption enables partially authorised brokers to implement content-based routing based on the attributes that are accessible to them resulting in more efficient event routing.

Our experiments show that (i) by caching plaintext and ciphertext content when possible, we are able to deliver comparable performance to plaintext events, and (ii) attribute level encryption incurs far less overhead than trying to emulate the same level of expressiveness with event level encryption.

By placing trust in some of the event brokers, namely the local brokers of both publishers and subscribers, we are able to implement both content-based routing and event encryption without having to rely on exotic cryptographic methods.

CHAPTER 8

Conclusions

This chapter concludes the dissertation. We summarise the contributions of this dissertation in §8.1; discuss the research avenues that have been revealed by our work in §8.2; and conclude the dissertation with §8.3.

8.1 Contributions

The goal for this work was to provide an access control architecture for decentralised, large-scale publish/subscribe systems that span multiple independent administrative domains, as described in Chapter 1. In Chapter 2 we provided an overview of previous research both in distributed communication paradigms as well as access control. In order to define the scope of our work we described in Chapter 3 what we understood to be a multi-domain publish/subscribe system, how we imagined one to be deployed, and what kind of threats we expected to see in such an environment.

In Chapter 4 we presented a scheme for securing event type definitions. The goal was to provide a scheme for globally unique names in a publish/subscribe context, so that those names could be used in an access control policy unambiguously to refer to resources. We also wanted to provide secure event type definitions that would allow users to verify their authenticity and integrity. Authenticity means that the event type definition is owned by a given principal. Integrity means that the event type definition has not been tampered with since it was deployed. The ownership of an event type is important, because it provides the root of the authority that is being delegated to domains and domain members with authorisation certificates. Type definition integrity is important also from an access control point of view, because it guarantees that all parties in the system agree on the contents of the type definition and can therefore enforce and

adhere to a common access control policy.

The work on secure event types also resulted in a scheme for evolving type definitions. We felt that one of the requirements for a large-scale publish/subscribe system is the ability of the system to evolve without it having to be shut down while doing this. Decentralised publish/subscribe systems usually allow the event service to evolve, because event brokers can be removed and added to the system at will in many cases, but there was no way to make changes to an event type that had already been deployed on the system without disconnecting existing clients. While providing support for secure event types, we also described a scheme for event type versioning that allows multiple versions of an event type to coexist on the same publish/subscribe system. The different versions would also share as much of the publication content as possible, i.e. a publication of version `1` would be translated to version `2` at the subscribers. This mechanism allows an event type to be used in a live system while publishers and subscribers slowly migrate to using the latest version of the event type definition.

In Chapter 5 we presented our access control architecture for decentralised, large-scale publish/subscribe systems. The architecture relied on SPKI authorisation certificates for delegating access rights to publish/subscribe principals. We concentrated especially on the multi-domain environment and therefore presented a hierarchical delegation model, where resource owners delegate access rights to domains and those domains delegate the access rights to sub-domains or event brokers and event clients. We felt that in a multi-domain environment the main goal should be to support scalability as much as possible. By treating event types as resources that are owned by their creators we are able to decentralise event type creation. By relying on SPKI authorisation certificates we are able to decentralise credential management and credential verification.

The access control architecture presented in Chapter 5 resulted in a system where access control towards event clients was enforced by the client's local event broker. The event brokers could be trusted to enforce the local domain's access control policy, but there was nothing to enforce access control policies in the broker network.

This dissertation concentrated on providing an access control mechanism for multi-domain publish/subscribe systems, but we felt that we should also address policy management, especially credential revocation, which is a hotly debated topic for capabilities. In Chapter 6 we presented a simple model for implementing policy management in domains with the OASIS RBAC system. We also described the various options for certificate revocation that are applicable to SPKI authorisation certificates. Finally we presented a scheme for implementing request/reply and reply-caching mechanisms in decentralised publish/subscribe system and how to use those mechanisms for delivering certificate validity statements to verifiers over a publish/subscribe system.

In Chapter 7 we presented an approach for encrypting event content. The goal was to prevent unauthorised event brokers from reading or writing event content by encrypting the events. This would allow the unauthorised brokers to route events in the event broker network. We

presented two schemes for event content encryption: whole event encryption and attribute level encryption. The former was more efficient, while the latter was more expressive.

8.2 Future Work

The work in this dissertation has highlighted a number of topics that should be addressed in future work.

We described in §6.4 how it would be beneficial if the verifier, i.e. the event client's local event broker, were to be able to evaluate predicates like the time of day or the publication frequency of the event client. Such an enhancement would allow the domain to issue certificates to the event clients that would be valid for longer periods of time, while providing the same level of control as short-lived certificates provide. For example, the verifier could enforce restrictions on the time of access instead of the domain having to issue short-lived certificates that expire every day at the time when the client is no longer authorised to access the event service.

PolicyMaker and other trust management systems support so called safe languages that are used to write short programs that either accept or reject an access request based on the local policy and the client's credentials (See §2.5 for a discussion on PolicyMaker). Safe in this case means that the program is guaranteed to finish and not to consume unreasonable amounts of resources. A similar language could be used in MAIA to move some of the dynamic parts of the access control policy to be enforced by the event brokers rather than the domain's access control service.

The SPKI-based access control model in MAIA allows for principals to have read access, write access, or both to an event type and its attributes. The scheme that is used to enforce access control within the event broker network by encrypting event content does not unfortunately provide the same level of granularity. That is, if an event broker has either read or write access to the event content, it will automatically be also able to write and read the content, respectively. This is because MAIA uses symmetric encryption where the same key is used to both encrypt and decrypt the event content. The obvious solution is to use asymmetric cryptography where the event content is encrypted with one key and decrypted with the other key. The brokers with read access to the event content would be issued one of the keys while those brokers with write access would be issued the other key. Public key as a term is in this case not appropriate, because both keys of the key pair would have to be treated as confidential information. An alternative approach would be to sign the event content with an asymmetric key and encrypt it with a symmetric key. Thereby only those event brokers that have access to the private key used to sign publications are able to publish events. The downside of any kind of asymmetric cryptography is that it is orders of magnitude more expensive than symmetric cryptography. Therefore we decided in this version of MAIA to not enforce access control at the level of different access operations in the event broker network.

Another aspect of security that is not addressed by the current architecture is the possibility of traffic analysis and event client privacy. In the current system event brokers are free to monitor the events that pass through them. While the events might be encrypted, some level of header information must be left in plaintext so as to enable more efficient event routing. Therefore intermediate brokers will be able to keep track of what types of events pass through them. On the other hand the event routing mechanism in MAIA is based on hashed event identifiers, i.e. $h(P||n)$, where P is the public key of the event type owner and n is the event type's human readable name. By using only this event type identifier in the event header we can *blind* the event type information so that a casual observer cannot know what event type it is. Authorised brokers can form a list of event type identifiers and compare events to that list when filtering them or delivering them to event clients. Unfortunately the number of event types in a system is not very large and it is conceivable that a dedicated adversary is able to get a list of most of the event type names in the system, e.g. from a type registry. The adversary can then calculate event type identifiers for all the event types on that list and compare the event type identifier of each passing publication to the list of event type identifiers, thereby being able to monitor the event flow in the system. We can defend against this attack by using a HMAC construct instead of a normal hash function to generate the event type identifier. The HMAC construct (i.e. *a keyed-hash message authentication code*) is defined in the IETF RFC 2104 [KBC97]. In simple terms, a secret key K is used in the hash calculation of the message m: $hmac(K, m)$. This prevents anyone who does not know the secret key from verifying the hash value and from generating new hash values. In our case this scheme will prevent unauthorised event brokers from creating lists of event type identifiers, because they are not in possession of the event type encryption key. The downside of this scheme is that the event type identifier would change whenever the encryption key is refreshed. It would be very difficult to somehow synchronise all brokers in the system to update the event type identifier at the same time without losing any publications in the process. We would like to see a scheme that would protect the system against these kinds of traffic monitoring attacks, even though one can argue that we should not be worried about such attacks given our threat model (See §3.6).

Raiciu and Rosenblum presented an alternative approach for implementing confidentiality in publish/subscribe systems in [RR06]. Most notably their approach relied on advanced cryptographic operations and allowed unauthorised brokers to route events based on their content without actually disclosing it. We assume that the broker network is trusted to some extent and that the decreased routing performance, in cases where the broker does not have access to the event content, is acceptable. It would be worthwhile to measure how much the routing performance in a large-scale system improves by allowing all nodes to route based on the event content, and on the other hand, what kind of an overhead is introduced by the advanced cryptographic operations.

Finally, we have realised that the research community would benefit from a modular publish/subscribe middleware that would allow for the mixing and matching of various publish/

subscribe models. Such a middleware would allow researchers to build on a common base, while being able to easily implement their own enhancement to the common middleware and to disable or enable other features like content-based routing and event encryption. Raiciu et al. suggested in their position paper that event matching and event routing should be separated in a publish/subscribe middleware in order to allow the middleware to cope with application diversity. We feel that a modular middleware implementation would cater for that need as well.

8.3 Summary

This dissertation has presented MAIA, a decentralised access control architecture for large-scale, multi-domain publish/subscribe systems. We have also addressed related issues like ambiguous names, forged event type definitions, and enforcing access control within the event broker network. We have motivated our work with a running example, based on the challenges facing the UK Police Forces and the Police Information Technology Organisation. Finally we have highlighted new research areas that we aim to explore in the future.

Bibliography

[AEM99] Marcel Altherr, Martin Erzberger, and Silvano Maffeis. iBus - a software bus middleware for the java platform. In *International Workshop on Reliable Middleware Systems*, pages 43–53, October 1999.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[And72] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, October 1972.

[Apa07] Apache activemq, October 2007. `http://activemq.apache.org/`.

[Aur99] Tuomas Aura. Distributed access-rights managements with delegations certificates. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 211–235. Springer-Verlag, 1999.

[BBHM95] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using events to build distributed applications. In *SDNE'95: Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments*, pages 148–155, Washington, DC, USA, June 1995. IEEE Computer Society.

[BCJ+90] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Keith Marzullo, Messac Makpangou, Ken Kane, Frank Schmuck, and Mark Wood. *The ISIS System Manual, Version 2.0*. Department of Computer Science, Cornell University, Ithaca, NY, USA, March 1990.

[BCM+99a] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS'99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–272, Washington, DC, USA, May 1999. IEEE Computer Society.

[BCM+99b] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. Information flow based event distribution middleware. In *Proceedings of the ICDCS Workshop on Electronic Commerce and Web-Based Applications*, pages 114–121, June 1999.

[Bel74] David E. Bell. Secure computer systems: A refinement of the mathematical model. Technical Report ESD-TR-73-278, Vol. III, Electronic Systems Division, Air Force Systems Command, April 1974.

[BEMP05] Jean Bacon, David M. Eyers, Ken Moody, and Lauri I. W. Pesonen. Securing publish/subscribe for multi-domain systems. In Gustavo Alonso, editor, *Middleware'05: Proceedings of the 6th International Conference on Middleware*, volume 3790 of *LNCS*, pages 1–20. Springer-Verlag, November 2005.

[BEP+03] András Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In *DEBS'03: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*, pages 1–8, New York, NY, USA, June 2003. ACM.

[BFIK99a] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis. The KeyNote trust-management system version 2. RFC 2704, Internet Engineering Task Force, September 1999.

[BFIK99b] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet programming: Security Issues for Mobile and Distributed Objects*, pages 185–210, London, UK, 1999. Springer-Verlag.

[BFK98] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). In *Proceedings of the Cambridge 1998 Security Protocols International Workshop*, volume 1550 of *LNCS*, pages 59–63, 1998.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the IEEE Conference on Security and Privacy*, pages 164–173. IEEE Computer Society, May 1996.

[Bib75] Ken Biba. Integrity considerations for secure computing systems. Technical Report MTR-3153, The MITRE Corporation, March 1975.

[BIK00] Matt Blaze, John Ioannidis, and Angelos Keromytis. DSA and RSA key and signature encoding for the KeyNote trust management system. RFC 2792, Internet Engineering Task Force, March 2000.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 311–322. ACM Press, 1987.

[BL73] David E. Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Vol. I, Electronic Systems Division, Air Force Systems Command, November 1973.

[BL76] David E. Bell and Leonard J. LaPadula. Secure computer system: Unified exposition and MULTICS intepretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, March 1976.

[BMY02] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4):492–540, November 2002.

[BMY03] Jean Bacon, Ken Moody, and Walt Yao. Access control and trust in the use of widely distributed services. *Software – Practice and Experience*, 33(4):375–394, 2003.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systystems*, 2(1):39–59, February 1984.

[BN89] David F. C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, pages 206–214, Oakland, CA, USA, May 1989. IEEE Computer Society.

[BRW03] Mihir Bellare, Phillip Rogaway, and David Wagner. EAX: A conventional authenticated-encryption mode. Cryptology ePrint Archive, Report 2003/069, April 2003. `http://eprint.iacr.org/2003/069`.

[Car98] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, December 1998.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2001.

[CFL+97] Yang-Hua Chu, Joan Feigenbaum, Brian A. LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[Cha93] David Chappell. *Distributed computing: implementation and management strategies*, chapter The OSF Distributed Computing Environment (DCE), pages 175–199. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, October 1993.

[CMPC04] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In *ICDCS'04: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, pages 552–561, Washington, DC, USA, March 2004. IEEE Computer Society.

[CNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[CRW99] Antonio Carzaniga, David R. Rosenblum, and Alexander L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *ICSE'99 Workshop on Engineering Distributed Objects (EDO '99)*, May 1999.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, August 2001.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Computer Society Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society, 1987.

[CW01] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, AZ, USA, October 2001. Springer-Verlag.

[DA99] Tim Dierks and Christopher Allen. The TLS protocol, version 1.0. RFC 2246, Internet Engineering Task Force, January 1999.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *POLICY'01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, January 2001. Springer-Verlag.

[DFFT02] Yanlei Diao, Peter Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and scalable filtering of xml documents. In *ICDE'02: Proceedings of the 18th International Conference on Data Engineering*, pages 341–342, Washington, DC, USA, February 2002. IEEE Computer Society.

[DH79] Whitfield Diffie and Martin Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67:397–427, March 1979.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[EFL+99] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylönen. SPKI certificate theory. RFC 2693, Internet Engineering Task Force, September 1999.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 254–269, New York, NY, USA, October 2001. ACM Press.

[Ell99] Carl Ellison. SPKI requirements. RFC 2692, Internet Engineering Task Force, September 1999.

[FGKZ03] Ludger Fiege, Felix C. Gärtner, Oliver Kasten, and Andreas Zeidler. Supporting mobility in content-based publish/subscribe middleware. volume 2672 of *LNCS*, pages 103–122. Springer-Verlag, June 2003.

[Fie00] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, CA, USA, 2000.

[FIP01] Specification for the advanced encryption standard (AES). Federal Information Processing Standards Publication 197, November 2001.

[FIP02] Secure hash standard (SHS). Federal Information Processing Standards Publication 180-2, August 2002.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, October 1992.

[FMB01] Ludger Fiege, Gero Mühl, and Alejandro Buchmann. An architectural framework for electronic commerce applications. In *Informatik 2001: Annual Conference of the German Computer Society*, 2001.

[Fre07] The Pastry web site, October 2007. `http://freepastry.rice.edu/`.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1985.

[Gon89] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, pages 56–63. IEEE Computer Society, May 1989.

[Hay96] Richard Hayton. *An Open Architecture for Secure Interworking Services*. PhD thesis, University of Cambridge, Cambridge, UK, 1996.

[Hoa78] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[IBM07] IBM WebSphere MQ. IBM Website, September 2007. `http://www.ibm.com/software/integration/wmq/`.

[IK03] Tetsu Iwata and Kaoru Kurosawa. OMAC: One-key CBC MAC. In *FSE'03: Fast Software Encryption*, volume 2887 of *LNCS*, pages 129–153. Springer-Verlag, February 2003.

[ISO98] ISO/IEC, Geneva, Switzerland. *9798-3:1998: Information Technology – Security Techniques – Entity Authentication – Part 3: Mechanisms Using Digital Signature Techniques*, 2 edition, 1998.

[ITU05a] ITU-T. *X.500: Information technology - Open Systems Interconnection - The Directory: Overview of concepts, models and services*, August 2005.

[ITU05b] ITU-T. *X.509: Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*, August 2005.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, Internet Engineering Task Force, February 1997.

[Khu05] Himanshu Khurana. Scalable security and accounting services for content-based publish/subscribe systems. In *SAC'05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 801–807, New York, NY, USA, March 2005. ACM.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC'97: Proceedings of the 29th annual ACM Symposium on Theory of computing*, pages 654–663, New York, NY, USA, May 1997. ACM Press.

[Lam74] Butler W. Lampson. Protection. *ACM SIGOPS Operating System Review*, 8(1):18–24, January 1974.

[LB73] Leonard J. LaPadula and David E. Bell. Secure computer systems: A mathematical model. Technical Report ESD-TR-73-278, Vol. II, Electronic Systems Division, Air Force Systems Command, November 1973.

[LRW00] Helger Lipmaa, Phillip Rogaway, and David Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, ML, USA, 2000.

[LS88] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI'88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 260–267, New York, NY, USA, June 1988. ACM.

[MB98] Chaoying Ma and Jean Bacon. COBEA: A CORBA-based event architecture. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 117–132, April 1998.

[Mik02] Zoltán Miklós. Towards an access control mechanism for wide-area publish/subscribe systems. In *DEBS'02: Proceedings of the 1st International Workshop on Distributed Event-based Systems*, July 2002.

[MM02] Peter Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *IPTPS'02: Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, pages 53–65, March 2002.

[MOV96] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, October 1996.

[MPR01] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A middleware for physical and logical mobility. In *ICDCS'01: Proceedings of the 21st*

IEEE International Conference on Distributed Computing Systems, page 524, Washington, DC, USA, April 2001. IEEE Computer Society.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[MUHW04] Gero Mühl, Andreas Ulbrich, Klaus Herrmann, and Torben Weis. Disseminating information to mobile clients using publish-subscribe. *IEEE Internet Computing*, 8(3):46–53, May 2004.

[New07] BBC News. Met given real time C-charge data. BBC News Website, July 2007. `http://news.bbc.co.uk/1/hi/uk_politics/6902543.stm`.

[Obj04a] The Object Management Group (OMG). *Common Object Request Broker Architecture: Core Specification, Revision 3.0.3*, March 2004.

[Obj04b] The Object Management Group (OMG). *CORBA Notification Service Specification, Revision 1.1*, October 2004.

[OP01] Lukasz Opyrchal and Atul Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, August 2001.

[Ope97] The Open Group. *DCE 1.1: Remote Procedure Call*, 1997.

[OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and Systems Security (TISSEC)*, 3(2):85–106, May 2000.

[PB02] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *DEBS'02: Proceedings of the 1st International Workshop on Distributed Event-Based Systems*, pages 611–618, Washington, DC, USA, July 2002. IEEE Computer Society.

[PB03] Peter R. Pietzuch and Jean Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *DEBS'03: Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, New York, NY, USA, June 2003. ACM Press.

[PB05] Lauri I. W. Pesonen and Jean Bacon. Secure event types in content-based, multi-domain publish/subscribe systems. In *SEM'05: Proceedings of the 5th international workshop on Software Engineering and Middleware*, pages 98–105. ACM Press, September 2005.

[PCM03] Gian Pietro Picco, Gianpaolo Cugola, and Amy L. Murphy. Efficient content-based event dispatching in the presence of topological reconfiguration. In *ICDCS'03: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 234–243, Washington, DC, USA, May 2003. IEEE Computer Society.

[PEB06] Lauri I. W. Pesonen, David M. Eyers, and Jean Bacon. A capabilities-based access control architecture for multi-domain publish/subscribe systems. In *SAINT 2006: Proceedings of the Symposium on Applications and the Internet*, pages 222–228, Washington, DC, USA, January 2006. IEEE Computer Society.

[PEB07] Lauri I.W. Pesonen, David M. Eyers, and Jean Bacon. Access control in decentralised publish/subscribe systems. *Journal of Networks*, 2(2):57–67, April 2007.

[Pie04] Peter R. Pietzuch. Hermes: A scalable event-based middleware. Technical Report UCAM-CL-TR-590, University of Cambridge, Computer Laboratory, June 2004.

[Pow96] David Powell. Group communication. *Communications of the ACM*, 39(4):50–53, April 1996.

[PR85] Jon Postel and Joyce Reynolds. File transfer protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985.

[RBBK01] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In *CCS'01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 196–205, New York, NY, USA, November 2001. ACM Press.

[RD01a] Antony Rowstron and Peter Druschel. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS-VIII: Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 75–80, Los Alamitos, CA, USA, May 2001. IEEE Computer Society.

[RD01b] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware'01: Proceedings of the 2nd International Conference on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 329–350, London, UK, November 2001. Springer-Verlag.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 161–172, New York, NY, USA, September 2001. ACM Press.

[RH03] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Computing Surveys*, 35(3):309–329, September 2003.

[Rin07] Rinda API, September 2007. `http://www.ruby-doc.org/stdlib/libdoc/rinda/rdoc/index.html`.

[Riv92] Ron Rivest. The MD5 message-digest algorithm. RFC 1321, Internet Engineering Task Force, April 1992.

[RL96] Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, October 1996.

[RR06] Costin Raiciu and David S. Rosenblum. Enabling confidentiality in content-based publish/subscribe infrastructures. In *Securecomm 2006: Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks*, August 2006.

[RW03] Phillip Rogaway and David Wagner. A critique of CCM. Cryptology ePrint Archive, Report 2003/070, April 2003. `http://eprint.iacr.org/2003/070`.

[SAS01] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness – transparent information delivery for mobile and invisible computing. In *CCGRID'01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, pages 277–285, Washington, DC, USA, May 2001. IEEE Computer Society.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SCR+03] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network file system (NFS) version 4 protocol. RFC 3530, Internet Engineering Task Force, April 2003.

[SL05] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with EventGuard. In *CCS'05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 289–298, New York, NY, USA, November 2005. ACM Press.

[SL07] Mudhakar Srivatsa and Ling Liu. Secure event dissemination in publish-subscribe networks. In *ICDCS'07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 22, Washington, DC, USA, June 2007. IEEE Computer Society.

[SM98] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *RBAC'98: Proceedings of the 3rd ACM Workshop on Role-Based Access Control*, pages 47–54, New York, NY, USA, October 1998. ACM Press.

[SM03] Alan T. Sherman and David A. McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering*, 29(5):444–458, May 2003.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001)*, pages 149–160, New York, NY, USA, September 2001. ACM Press.

[Sri95] Raj Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Internet Engineering Task Force, August 1995.

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63:1278–1308, September 1975.

[Sto07] Stomp messaging protocol, October 2007. `http://stomp.codehaus.org/`.

[Sun88] *RPC: Remote Procedure Call Protocol Specification Version 2*, June 1988.

[Sun94] Java™remote method invocation (RMI), 1994. `http://java.sun.com/rmi/`.

[Sun99] Sun Microsystems, Inc. *Code Conventions for the Java™Programming Language*, April 1999.

[Sun02] Sun Microsystems, Inc. *Java Message Service, Version 1.1*, 2002. `http://java.sun.com/products/jms/`.

[Sun03] Sun Microsystems, Inc. *JavaSpaces™Service Specification, Version 2.0*, June 2003.

[TIB07] Tibco rendezvous, September 2007. `http://www.tibco.com/`.

[Tra07] The Transport for London congestion charge website, September 2007. `http://www.cclondon.com/`.

[WABL94] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, February 1994.

[WCEW02] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L. Wolf. Security issues and requirements in internet-scale publish-subscribe systems. In *HICSS'02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, page 303, Washington, DC, USA, January 2002. IEEE Computer Society.

[WGL00] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Netwworking (TON)*, 8(1):16–30, February 2000.

[WHF03] Doug Whiting, Russ Housley, and Niels Ferguson. Counter with CBC-MAC (CCM). RFC 3610, Internet Engineering Task Force, September 2003.

[Whi76] James E. White. A high-level framework for network-based resource sharing. RFC 707, Internet Engineering Task Force, January 1976.

[Wil79] Maurice V. Wilkes. *The Cambridge CAP Computer and its Operating System (Operating and Programming Systems Series)*. North Holland, Amsterdam, The Netherlands, 1979.

[Win99] Dave Winer. XML-RPC specification. XML-RPC Website, June 1999. `http://www.xmlrpc.com/spec`.

[WMLF98] Peter Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[Wor05] World Wide Web Consortium. *XML Path Language (XPath) Version 2.0*, April 2005.

[Wor07] World Wide Web Consortium. *SOAP Version 1.2*, April 2007. `http://www.w3.org/TR/soap12-part1/`.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories Inc., November 1994.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. volume 3494 of *LNCS*, pages 19–35. Springer, May 2005.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision search attacks on SHA1. Technical report, Shandong University, Shandong, China, 2005.

[XL89] Andrew Xu and Barbara Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Digest of Papers of the 19th International Symposium on Fault-Tolerant Computing*, pages 199–206, Chicago, IL, USA, June 1989. IEEE Computer Society.

[ZF03] Andreas Zeidler and Ludger Fiege. Mobility support with REBECA. In *ICDCSW'03: Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, pages 354–360, Providence, RI, USA, May 2003. IEEE Computer Society.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[ZS06] Yuanyuan Zhao and Daniel C. Sturman. Dynamic access control in a content-based publish/subscribe system with delivery guarantees. In *ICDCS'06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 60, Los Alamitos, CA, USA, July 2006. IEEE Computer Society.