Technical Report

UCAM-CL-TR-590
ISSN 1476-2986

Number 590

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Hermes: A scalable event-based middleware

Peter R. Pietzuch

June 2004

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 February 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Queens' College.

ISSN 1476-2986

Abstract

Large-scale distributed systems require new middleware paradigms that do not suffer from the limitations of traditional request/reply middleware. These limitations include tight coupling between components, a lack of information filtering capabilities, and support for one-to-one communication semantics only. We argue that event-based middleware is a scalable and powerful new type of middleware for building large-scale distributed systems. However, it is important that an event-based middleware platform includes all the standard functionality that an application programmer expects from middleware.

In this thesis we describe the design and implementation of HERMES, a distributed, event-based middleware platform. The power and flexibility of HERMES is illustrated throughout for two application domains: Internet-wide news distribution and a sensor-rich, active building. HERMES follows a type- and attribute-based publish/subscribe model that places particular emphasis on programming language integration by supporting type-checking of event data and event type inheritance. To handle dynamic, large-scale environments, HERMES uses peer-to-peer techniques for autonomic management of its overlay network of event brokers and for scalable event dissemination. Its routing algorithms, implemented on top of a distributed hash table, use rendezvous nodes to reduce routing state in the system, and include fault-tolerance features for repairing event dissemination trees. All this is achieved without compromising scalability and efficiency, as is shown by a simulational evaluation of HERMES routing.

The core functionality of an event-based middleware is extended with three higher-level middleware services that address different requirements in a distributed computing environment. We introduce a novel congestion control service that avoids congestion in the overlay broker network during normal operation and recovery after failure, and therefore enables a resource-efficient deployment of the middleware. The expressiveness of subscriptions in the event-based middleware is enhanced with a composite event service that performs the distributed detection of complex event patterns, thus taking the burden away from clients. Finally, a security service adds access control to HERMES according to a secure publish/subscribe model. This model supports fine-grained access control decisions so that separate trust domains can share the same overlay broker network.

To my parents

Declaration

This dissertation is not substantially the same as any that I have submitted or am currently submitting for a degree, diploma or any other qualification at any other university.

No part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or any other qualification.

The work for Chapter 6 was done with Sumeer Bhola while at the IBM TJ Watson Research Center [PB03b]. Material from Chapter 7 is based on a paper co-authored with Brian Shand [PSB03]. Chapter 8 is based on work with András Belokosztolszki, David Eyers, and Brian Shand [BEP$^+$03].

This dissertation is the result of my own work and is not the outcome of work done in collaboration, except as specified in the text.

This dissertation does not exceed $60,000$ words, including tables and footnotes, but excluding bibliography and diagrams.

Acknowledgements

First of all, I would like to thank Jean Bacon, my supervisor, who has provided tremendous advice and support during the time of my PhD. Her insight helped my work go in the right direction, and she taught me the principles of research work. I am extremely grateful for this. I also appreciate the assistance and feedback given by Ken Moody.

I have had the pleasure of being a member of the Opera Group. In particular, I enjoyed collaborating with András Belokosztolszki, David Eyers, and Brian Shand on various topics. Many ideas stem from (often hour-long) tea-breaks with the regular tea crowd that included Alan, Alexis, András, Aras, Brian, Chaoying, Chris, Dan, David, Eiko, Lauri, Nathan, Niki, Salman, and Walt. I would also like to thank Alan Abrahams, David Eyers, and András Belokosztolszki for proof-reading my thesis and suggesting improvements.

I am grateful to Sumeer Bhola, who was my mentor while I was doing an internship at the IBM TJ Watson Research Center. His guidance led to some of the work described in this thesis.

My research was funded by the Engineering and Physical Sciences Research Council (EPSRC), the Cambridge European Trust, and QinetiQ, Malvern.

Finally, I would like to thank my parents for their support and encouragement throughout the years, which made it all possible.

Publications

[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 H. Arno Jacobsen, editor, *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003. ACM.

[CBP$^+$02] Jon Crowcroft, Jean Bacon, Peter Pietzuch, George Coulouris, and Hani Naguib. Channel Islands in a Reflective Ocean: Large-scale Event Distribution in Heterogeneous Networks. *IEEE Communications Magazine*, 40(9):112–115, September 2002.

[KDHP03] Evangelos Kotsovinos, Boris Dragovic, Steven Hand, and Peter R. Pietzuch. XenoTrust: Event-Based Distributed Trust Management. In *Proceedings of Trust and Privacy in Digital Business (TrustBus'03). In conjunction with the 14th International Conference on Database and Expert Systems Applications (DEXA'03)*, Prague, Czech Republic, September 2003.

[PB02] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In Jean Bacon, Ludger Fiege, Rachid Guerraoui, H. Arno Jacobsen, and Gero Mühl, editors, *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02). In conjunction with the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 611–618, Vienna, Austria, July 2002. IEEE.

[PSB03] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A Framework for Event Composition in Distributed Systems. In Markus Endler and Douglas Schmidt, editors, *Proceedings of the 4th International Conference on Middleware (Middleware'03)*, volume 2672 of *LNCS*, pages 62–82, Rio de Janeiro, Brazil, June 2003. **Best Paper Award**. ACM/IFIP/USENIX, Springer Verlag.

[PB03a] Peter R. Pietzuch and Jean Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In H. Arno Jacobsen, editor, *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003. ACM.

[PB03b] Peter R. Pietzuch and Sumeer Bhola. Congestion Control in a Reliable Scalable Message-Oriented Middleware. In Markus Endler and Douglas Schmidt, editors, *Proceedings of the 4th International Conference on Middleware (Middleware'03)*, volume 2672 of *LNCS*, pages 202–221, Rio de Janeiro, Brazil, June 2003. ACM/IFIP/USENIX, Springer Verlag.

[PSB04] Peter R. Pietzuch, Brian Shand, and Jean Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, 18(1):44–55, January/February 2004.

Contents

List of Figures

List of Tables

1 Introduction

Event-based middleware is a scalable and powerful new middleware paradigm for building large-scale distributed systems. This thesis proposes a fully-fledged event-based middleware, as opposed to a simple publish/subscribe system for message communication, that provides advanced features, such as type- and attribute-based event routing, programming language integration, and common higher-level middleware services, and all without compromising efficiency. Overlay routing techniques, as found in many peer-to-peer systems, play an important role when building a scalable event-based middleware for Internet-scale applications. Higher-level middleware services, such as the detection of composite event patterns, the avoidance of congestion in the overlay network, and the encryption of event messages, enhance the expressiveness and usability of the event-based middleware to a distributed applications programmer. Thus, an event-based middleware can be considered the next generation of middleware, for it will be a fundamental building block for future complex distributed applications that span the Internet.

1.1 Large-Scale Distributed Systems

With the advent of the Internet, it became possible to build large-scale distributed applications because of the existence of a global, packet-based communication infrastructure. This increase in application scale by several orders of magnitude results in systems with millions of nodes that need to communicate and cooperate in order to achieve a common goal. We envision a world with large-scale e-commerce and business applications with many components operating over the Internet in a highly heterogeneous environment. The complexity of designing and building such applications can be high, which makes the use of a powerful middleware abstraction important. However, traditional middleware concepts that were mostly developed in the context of local area networks (LANs), do not perform well in an environment with wide area networks (WANs). As a result, new types of middleware are needed for today's large-scale distributed systems.

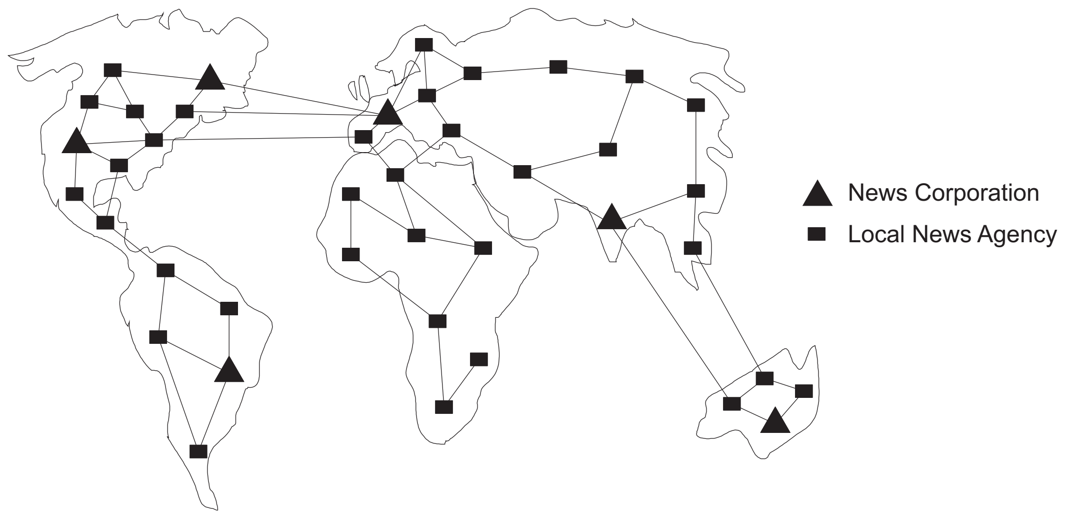


Figure 1.1: A news story dissemination system

1.2 Application Scenarios

In order to understand the requirements of a middleware for large-scale distributed systems, it is necessary to consider application scenarios, in which the use of traditional middleware would be prohibitively expensive in terms of efficiency or usability. In the following, we shall present two example applications for motivating our design of an event-based middleware throughout the rest of the dissertation. In general, large scale in an application may manifest itself by either a geographic distribution of nodes, when a global application is spread out over several continents, or by the concentration of nodes with high density on a relatively small area, such as in a ubiquitous computing environment within a single building.

1.2.1 Internet-Wide Distributed Systems

Internet-wide distributed systems often involve the exchange of information among a large number of nodes. A system for *news story dissemination* is depicted in Figure 1.1, where news reports that are generated by local news agencies are distributed world-wide among many news corporations. News corporations prefer to only receive information that they are interested in and news agencies do not want to deal with the complexity of having knowledge about all news corporations. In addition, the wide-area communication links between continents need to be managed efficiently so that congestion does not occur.

1.2.2 Large-Scale Ubiquitous Systems

A different type of large-scale distributed system is a ubiquitous sensor-rich environment concentrated on a small area, such as the *Active Office* building shown in Figure 1.2. In this building, sensors that are installed in offices provide information about the environment to interested devices, applications, and users. The Active Office is aware of its inhabitants' behaviour and enables them to interact with it in a natural way. The large number of sensors potentially produce a vast amount of data. Information consumers prefer a high-level view of the primitive

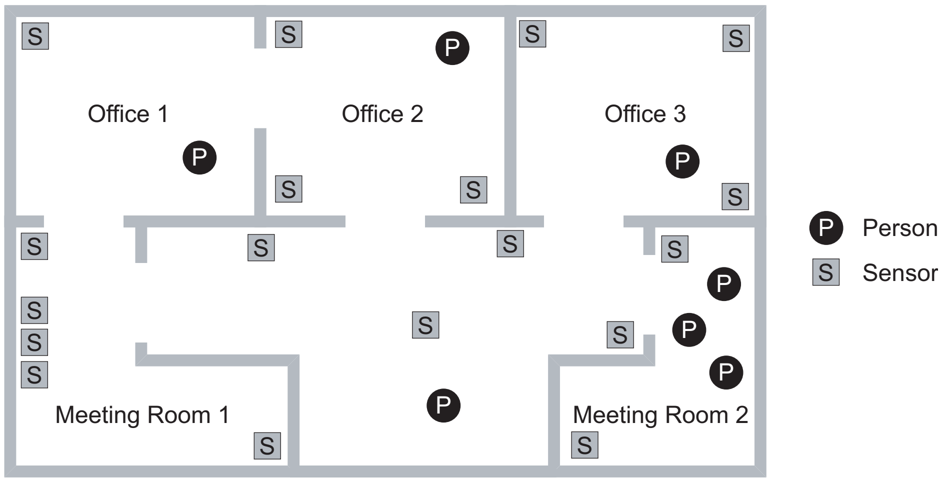


Figure 1.2: The Active Office ubiquitous environment

sensor data. Thus, a middleware used in this application scenario has to cope with high-volume data and be able to aggregate and transform it before dissemination.

1.3 Why Event-Based Middleware?

The concept of a *middleware* was introduced to facilitate communication between entities in a heterogeneous distributed computing environment. Middleware is usually an additional layer between the operating system and the distributed application on every node that deals with communication issues and attempts to provide a homogeneous view of the world to the application. As such, it is widely used and has proved to be a successful abstraction that helps with the design and implementation of complex distributed systems. However, traditional middleware is based on the idea of *request/reply communication* between two entities in the system. A client requests information from a server and then waits until the server has responded with a reply. The client object must know the identity of the server and can only interact with a single server at a time, which limits scalability.

Publish/subscribe communication is a more scalable paradigm that addresses the shortcomings of request/reply communication by supporting many-to-many interaction among entities. A client can be an information producer or an information consumer (see Figure 1.3). Consumers state what kind of information they are interested in receiving and it is then the responsibility of the publish/subscribe system to deliver this information from producers to consumers. This

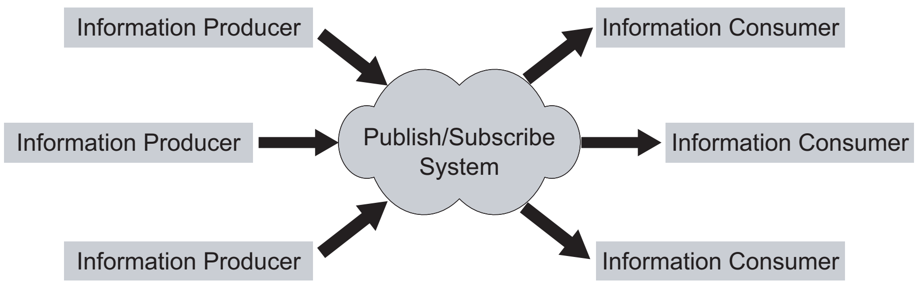


Figure 1.3: A publish/subscribe system

means that information producers and consumers do not need to know about each other and a single producer may cause information to be delivered to multiple consumers. It is obvious that this communication model fits the two application scenarios introduced in the previous section, as in both cases information has to be disseminated to all interested parties in the system.

A problem is that a pure publish/subscribe system is of limited use to a distributed applications programmer because the implementation of the communication model alone does not constitute an entire middleware platform. A better solution is to provide an *event-based middleware* that complements the scalable and efficient implementation of the publish/subscribe model with additional functionality, commonly found in traditional middleware, such as programming language integration or congestion control, and other functionality, such as composite pattern detection, which is specific to the publish/subscribe style of communication. Only when an event-based middleware supports all the features that a programmer expects from a middleware platform, will publish/subscribe communication be widely adopted for the engineering of large-scale distributed systems.

1.4 Research Statement

This thesis argues that event-based middleware will form the next generation of middleware for large-scale distributed systems. We make a case for our argument by describing a feature-complete event-based middleware called Hermes and proposing three higher-level middleware services that are implemented on top of Hermes to enhance its functionality. A novel model of an event-based middleware is introduced that derives from the requirements of large-scale distributed systems.

The main contribution of this work is the design and implementation of Hermes, a distributed, content-based, event-based middleware architecture that focuses on scalability, expressiveness, robustness, extensibility, usability, and interoperability. It substantially differs from previous publish/subscribe systems by combining ideas from publish/subscribe, peer-to-peer, database, middleware, and networking research. The main distinguishing features of Hermes are:

Peer-to-Peer Routing. The core content-based publish/subscribe functionality of Hermes is implemented on top of a distributed hash table and uses a novel content-based event routing algorithm to achieve scalable, robust, and efficient event dissemination. Peer-to-peer techniques are used to manage the overlay network of event brokers in Hermes and to recover from faults.

Simulational Evaluation. The efficiency of the Hermes' routing algorithm is evaluated in a distributed systems simulator. The performance is compared to a standard algorithm for content-based event routing in order to prove the scalability and efficiency claims of our approach.

Programming Language Integration. Hermes integrates publish/subscribe concepts into the programming language by supporting strong event typing and event binding. Nevertheless, Hermes is not tied to a single programming language, and maintains a language-independent representation of data.

Moreover, we develop three novel extensions to the core functionality of an event-based middleware. These higher-level middleware services address various requirements in large-scale distributed systems and are implemented on top of an event-based middleware such as Hermes.

Congestion Control. A congestion control algorithm is proposed that detects congestion in the overlay network of a content-based publish/subscribe system and removes its cause by modifying system parameters. The performance of the congestion control algorithm is evaluated with realistic experiments in an industrial-strength content-based publish/-subscribe system.

Composite Event Detection. The expressiveness of the event-based middleware is increased with a generic service for the distributed detection of composite event patterns. The composite event detection service is based on extended finite state automata and supports the decomposition of complex composite event expressions for distribution.

Security. We introduce a secure publish/subscribe model that adds access control to an event-based middleware. The model supports fine-grained access control decisions down to the level of individual event attributes and deals with various degrees of event broker trust.

1.5 Dissertation Outline

The remainder of this dissertation is organised as follows:

Chapter 2 provides a survey of the background that is necessary to understand the notion of an event-based middleware. It first considers various types of existing synchronous and asynchronous middleware and then gives an overview of current content-based publish/subscribe systems. The chapter finishes by looking at different types of large-scale peer-to-peer systems.

Chapter 3 considers the concept of an event-based middleware. It starts by identifying the requirements of an event-based middleware with respect to building large-scale distributed systems. From that, several models are derived that describe different aspects of an event-based middleware and map the design space.

Chapter 4 presents our incarnation of the event-based middleware model called HERMES. It starts with an architectural overview and then focuses on a detailed presentation of the peer-to-peer routing algorithms. The chapter concludes with a description of the HERMES implementation, emphasising language integration issues.

Chapter 5 evaluates the HERMES routing algorithms. After describing the simulational environment and experimental setup, it shows the results of four experiments comparing HERMES routing to a standard approach and discusses them.

Chapter 6 considers congestion in an event-based middleware and introduces our congestion control algorithms as the first middleware extension. It begins by analysing congestion in a content-based publish/subscribe system and then derives two complementary mechanisms to deal with it. The chapter finishes with an evaluation of the congestion control algorithms in experiments and a survey of relevant related work.

Chapter 7 presents a service for composite event detection, which is the second middleware extension. First, composite events and suitable application scenarios are introduced in more detail. After that, the chapter proposes a design and architecture for the distributed detection of composite events focusing on the composite event language and the composite event detectors. Finally, we deal with issues related to distributed detection and finish with an evaluation and related work.

Chapter 8 describes the third middleware extension that addresses security in an event-based middleware. The chapter analyses the requirements of access control in a content-based publish/-subscribe system by looking at application scenarios. It explains our secure publish/subscribe model focusing on boundary restrictions, event broker trust, and event attribute encryption. Furthermore, we discuss our prototype implementation, an evaluation, and related work.

Chapter 9 gives a brief conclusion, summarising the work described in this thesis, and outlines future work.

2 Background

The purpose of this chapter is to provide the necessary background required to understand the concepts that relate to an event-based middleware. When designing any novel type of middleware, it is important to learn from past research experience, which has resulted in many contrasting middleware technologies with different strengths and weaknesses. In addition, the Internet has led to increased research into peer-to-peer systems, which created algorithms and techniques for the design and implementation of large-scale distributed systems. Event-based middleware can be seen as the intersection of publish/subscribe systems with middleware technology and peer-to-peer systems.

Consequently, this chapter is structured in a similar fashion starting with an overview of current middleware technology in Section 2.1, discussing the applicability of each type of middleware to large-scale distributed systems. After that, state-of-the-art publish/subscribe systems are reviewed in Section 2.2 with special attention to the existence or lack of middleware features in more expressive content-based publish/subscribe. The chapter ends with Section 2.3 that describes different kinds of peer-to-peer systems that handle routing and information dissemination in large-scale distributed systems.

2.1 Middleware

The idea of a middleware first came up in the 1980s when LANs became commonplace. The high bandwidth connectivity of LANs made it simpler to build distributed systems consisting of multiple software components running on separate computers, however programmers had to explicitly deal with the heterogeneity of these systems due to different platforms, protocols, and programming languages. An obvious solution to the problem was a middleware layer between the operating system and the application that provides a uniform abstraction to all applications in the distributed computing environment [BH03]. Since there is no single agreed upon definition of middleware, it is difficult to state where the operating systems ends and the middleware begins. For the purpose of this dissertation, we will use a broad definition of middleware.

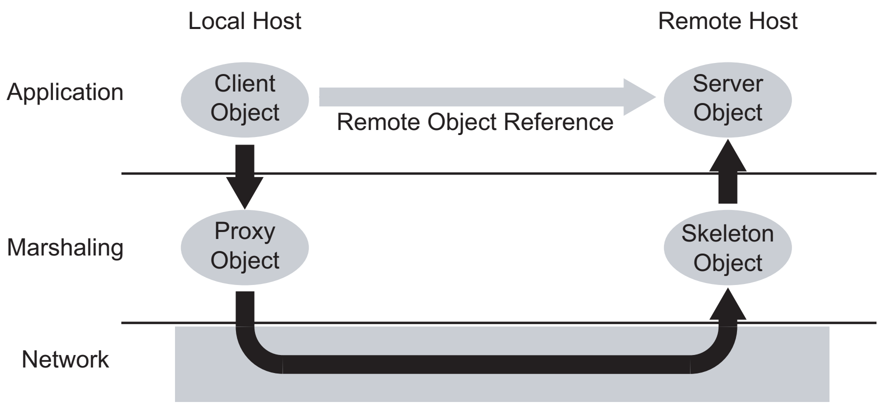


Figure 2.1: Components of a synchronous request/reply middleware

Definition 2.1 (Middleware) *A* middleware *is a software layer present on every node of a distributed system that uses operating system functions to provide a homogeneous high-level interface to applications for many aspects of a distributed computing environment, such as communication, naming, concurrency, synchronisation, replication, persistency, and access control.*

One of the main responsibilities of a middleware is to manage the communication of components in the distributed system. Thus, middleware can often be classified according to its communication model. A popular model for middleware is request/reply communication that derives from the programming language concept of a procedure call. In the next section, we will give an overview of *synchronous request/reply middleware* that follows this approach. Historically, the first type of middleware was *asynchronous messaging middleware* that uses messages to pass data between components. This type of middleware is more suited to large-scale systems, as will be shown in Section 2.1.2.

2.1.1 Synchronous Request/Reply Middleware

From an applications programmer's perspective, it is desirable to have to deal as little as possible with the extra complexity introduced by distribution. This led to the extension of the programming language concept of a local procedure or function call to a *remote procedure call* (RPC). In RPC, a function calls another function that is located on a different node in the system. Since today's widely-used programming languages are mostly object-oriented, in practice, methods on remote objects are invoked instead of single functions. A method of a *client object* invokes a method of a remote *server object* that runs on a different node, resulting in a client/server relation. Analogous to a local method call, the call is synchronous because the client is blocked until the remote method returns with a result value from the server.

Synchronous object-oriented request/reply middleware implements this communication abstraction as illustrated in Figure 2.1. An object reference can either point to a local or a remote object. A method invocation to a remote object is intercepted locally and the associated method parameters are marshaled into a representation that can be transfered over the network. The marshaling code is usually statically auto-generated by the middleware. After transmission, the parameters are unmarshaled again and the method of the remote object is invoked at the server-side. The whole process is repeated to return the result value from the server method back to the client.

Service Name	Description
Collection Service	enables objects to be grouped into collections
Event Service	provides asynchronous communication between objects
Naming Service	translates names into remote object references
Notification Service	extends the event service with filtered communication
Persistence Service	supports the storage of objects on disk
Security Service	deals with authentication and access control
Trading Services	allows clients to discover interfaces
Transaction Service	supports transactions across remote method calls

Table 2.1: Some object services in CORBA

The distribution transparency provided by a request/reply middleware creates the illusion that local and remote method calls can be treated uniformly. As a consequence, a *tight coupling* between clients and servers is introduced. Although this works well in a LAN, where network communication is reliable and inexpensive in terms of latency and bandwidth, the paradigm breaks down in WANs, where failure is common and the latency of remote calls can be several orders of magnitude higher than that of local ones. Another problem when building a large-scale distributed system with a request/reply middleware is that all communication is one-to-one between exactly two entities, the client and the server. By contrast, in a large-scale system, a single component may need to communicate with several other components in multicast style, potentially without even having knowledge of all its communication partners.

In the following, we will describe two commonly used synchronous middleware platforms, CORBA and Java RMI, in more detail. Even though the underlying synchronous communication paradigm is not well suited to large-scale systems, it is worthwhile to look at the additional services that are provided by those middleware platforms to ease the task of the application programmer. Moreover, many synchronous middleware platforms come with asynchronous extensions to alleviate the problem of tight coupling of communicating entities.

CORBA

The *Common Object Request Broker Architecture* (CORBA) [OMG02b] is a platform- and language-independent object-oriented middleware architecture designed by the Object Management Group (OMG). CORBA is released as a specification of an open standard that vendors can implement, which facilitates interoperability between different implementations. It is a mature middleware technology that is widely used in financial and telecommunication systems and has inspired many recent middleware initiatives. Some reasons for CORBA's success are its good programming language integration across several mainstream languages, the simple extensibility of the platform using object services, and its adaptation to heterogeneous distributed systems. The main parts of the CORBA framework are:

Object Request Broker (ORB). The ORB forms the core of the middleware and handles communication. It resolves object references, which are specified in the *Interoperable Object Reference* (IOR) format, to locations and includes an object adapter, which is responsible for interfacing with local implementations of server objects. Moreover, the ORB performs the marshaling and unmarshaling of method parameters and sends invocations over the network using the *General Inter-ORB Protocol* (GIOP).

Object Model. A powerful feature of CORBA is that it comes with its own object model that has primitive and object types to define interfaces of remote objects. The object model is language-independent and the specification provides mappings from the object model to type systems of common programming languages.

Object Definition Language (IDL). Interfaces of remote objects are defined in IDL, which is language-independent and supports various bindings. An IDL compiler transforms the static interface definitions into stub and skeleton source code in a given programming language that does the marshaling and unmarshaling of method arguments. This has the advantage that remote method invocations can be statically type-checked by the compiler.

Dynamic Invocation Interface (DII). If the interface of the remote object to be invoked is not known at compile-time, the DII allows the remote call to be constructed at runtime. Dynamic invocation is essential when implementing general purpose proxy or browser objects.

Interface and Implementation Repositories. The interface repository contains the IDL definitions of interfaces for type-checking remote method calls. The repository can be queried either at compile-time or at runtime. Correspondingly, the implementation repository contains all implementations of a remote interface at the server-side so that remote objects can be activated on demand.

Object Services. The CORBA platform is extensible by means of object services that address different facets of a distributed computing environment, ranging from transactional support to security. Table 2.1 lists a selection of object services.

Asynchronous Method Invocation (AMI). Standard CORBA provides *one-way* call semantics for best-effort method invocations that do not expect a return value and thus does not require blocking. A more advanced scheme is AMI where the result value of a remote method call is supplied in an asynchronous callback to the client.

Although CORBA tries to address the need for asynchronicity in middleware communication for large-scale systems, its core design is still based on synchronous communication. AMI is a step in the right direction but it is not a complete solution as it relies on extensions to the IDL that are hard to use. In addition, CORBA objects are heavy-weight entities due to their support for many CORBA features and are thus not suitable for a light-weight messaging protocol. An even more fundamental problem is that many-to-many communication is not part of the basic services provided by the ORB and can only be simulated by less efficient object services.

Nevertheless, the language-independent and extensible design of CORBA is an important lesson in how to build middleware for heterogeneous systems. In the next section, we will look at the CORBA Event and Notification Services that explicitly deal with asynchronous communication.

CORBA Event Service. The OMG acknowledged the need for publish/subscribe communication in a middleware by introducing the *CORBA Event Service* [OMG95] as a CORBA extension. It is centred around the concept of an *event channel*, which connects suppliers to consumers of data in a decoupled fashion. The communication among suppliers and consumers can be in *pull mode*, in which case a consumer requests data from suppliers via the event channel. The alternative is that suppliers push data to the consumers in *push mode*. Data is represented as *events* that are either typed or untyped. Untyped events are attributes of the CORBA datatype `any` that can be cast to any datatype. For typed communication, suppliers and consumers agree on a particular IDL interface and use its methods to exchange information in pull or push mode.

The Event Service enables CORBA clients to participate in many-to-many communication through an event channel. The asynchronous communication is implemented on top of CORBA's synchronous method invocation and thus has the substantial overhead of performing a remote method invocation for every event communication. Moreover, event consumers cannot limit the events that they receive from an event channel because no event filtering is supported.

CORBA Notification Service. The *CORBA Notification Service* [OMG02a] addresses the shortcomings of the Event Service by providing event filtering, quality of service (QoS), and a lightweight form of typed events. These *structured events* are divided into a header and a body. Both contain attribute/value pairs that hold the data associated with the event, but only the header attributes are externally visible. This allows event consumers to restrict the events that they receive from the event channel by specifying filters over the attributes using a *filter constraints language*. In addition, events are categorised according to their type with a *type name* field. A particular event may be uniquely identified through its *event name*.

With structured events, the Notification Service implements a sophisticated type mechanism for event data. It comes with an expressive content-based filtering language for events and has support for QoS attributes. However, since the Notification Service attempts to maintain backwards compatibility with the Event Service, it suffers from the same problems with regard to communication efficiency over synchronous request/reply.

Java RMI

The Java programming language is popular for network programming and has therefore built in middleware functionality. The *Java Remote Method Invocation* (RMI) specification [Sun99b] describes how to synchronously invoke methods of remote objects using request/reply communication between two *Java Virtual Machines* (JVMs) running on separate nodes. The Java RMI compiler generates marshaling code for the proxy object and the server skeleton. Because of the homogeneous environment created by JVMs, in which there is only a single programming language, the burden on the middleware is lower. It is even possible to move executable code between JVMs by using Java's *object serialisation* to flatten an object implementation into a byte stream for network transport.

Asynchronous event communication within a single JVM is mainly used in the *Abstract Window Toolkit* (AWT) libraries for graphical user interfaces. The `EventListener` interface can be implemented by a class to become a callback object for asynchronous events, such as mouse or keyboard events. The Jini framework, described below, extends this to provide event communication between JVMs. Other variants of asynchronous communication in Java are provided by the messaging infrastructure of JMS (Section 2.1.2) and the tuple spaces in the JavaSpaces specification (Section 2.1.3).

Jini. The *Jini* specification [Sun03b] enables programmers to create network-centric services by defining common functionality for service descriptions to be announced and discovered. For this, it supports distributed events between JVMs. A `RemoteEventListener` interface is capable of receiving remote callbacks of instances of the `RemoteEvent` class. A `RemoteEvent` object contains a reference to the Java object where the event occurred and an `eventID` that identifies the type of event. A `RemoteEventGenerator` accepts registrations from objects and returns instances of the `EventRegistration` class to keep track of registrations. It then sends

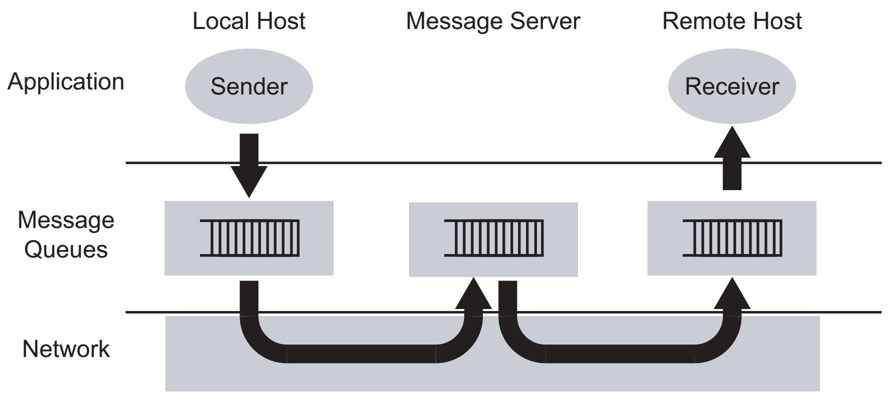


Figure 2.2: Components of an asynchronous messaging middleware

`RemoteEvent` objects to all interested `RemoteEventListeners`. Event generators and listeners can be decoupled by third-party agents, for example, to filter events, but the implementation is outside the Jini specification and left to the programmer.

As is the case for CORBA, event communication in Jini is built on top of synchronous Java RMI, so the same restrictions that limit scalability and efficiency apply. More recently, Jini started to use tuple spaces for asynchronous communication (Section 2.1.3).

2.1.2 Asynchronous Message-Oriented Middleware

A purely asynchronous approach to communication in a middleware is taken by a *message-oriented middleware* (MOM) [BCSS99]. Traditionally, it is used in connection with databases and transaction processing systems. Message-oriented middleware is based on the model of *message-passing* between a sender and a receiver in a distributed system, which is shown in Figure 2.2. A sender sends a message to a receiver by first inserting it into a local *message queue*. It then becomes the responsibility of the message-oriented middleware to deliver the message through the network to the receiver's message queue, which gets an asynchronous callback once the message has arrived. The sender and the receiver are *loosely-coupled* during their communication because they only communicate via the indirection of message queues and thus do not even have to be running at the same time. Message queues can provide strong reliability guarantees in case of failure by storing *persistent messages* on disk.

In a standard message-oriented middleware, communication between clients is still one-to-one but the model is easily extensible to have many-to-many semantics, such as in publish/subscribe. To increase scalability for large-scale systems, message queues can be stored on dedicated *message servers* that may transform messages in transit and support multi-hop routing of messages. A disadvantage of the message-passing model is that it is harder to integrate with a programming language because interaction with the middleware happens through external application programming interface (API) calls and is not part of the language itself, as is the case for remote method invocation. This means that messages cannot be statically type-checked by the compiler, but some messaging middlewares support dynamic type-checking of message content.

Event-based middleware and message-oriented middleware have many features in common because message-oriented middleware often includes publish/subscribe functionality. Therefore, we will look at JMS and IBM WebSphere MQ, two widely-used messaging platforms, to point out the characteristics of this type of middleware.

JMS

The *Java Message Service* (JMS) [Sun01] defines a messaging API for Java. JMS clients can choose any vendor-specific implementation of the JMS specification, called a JMS provider. JMS comes with two communication modes: point-to-point and publish/subscribe communication. *Point-to-point communication* follows the one-to-one communication abstraction of message queues. Queues are stored and managed at a *JMS server* that decouples clients from each other. Direct communication between a sender and a receiver without an intermediate server is not supported. In *publish/subscribe communication*, the JMS server manages a number of *topics*. Clients can publish messages to a topic and subscribe to messages from a topic.

Like structured CORBA events, a JMS message is divided into a message header and body. The header contains various fields, including the destination of the message, its delivery mode, a message identifier, the message priority, a type field, and a timestamp. The delivery mode can be set to `PERSISTENT` to enforce exactly-once delivery semantics, otherwise best-effort delivery applies. The type of a message is an optional field used by a JMS provider for type-checking the message. However, most JMS providers currently do not support this. Apart from predefined fields, the header can also contain any number of user-supplied fields. The message body is in one of several formats: a `StreamMessage`, a `TextMessage`, and a `ByteMessage` contain the corresponding Java primitive types. A `MapMessage` is a dictionary of name/value pairs similar to the fields found in the header. Finally, an `ObjectMessage` uses Java's object serialisation feature to transmit entire objects between clients.

JMS provides a topic-based publish/subscribe service with limited content-based filtering support in the form of *message selectors*. A message selector allows a client to specify the messages it is interested in by stating a filter on the fields in the message header. The selector syntax is based on a subset of the SQL92 [SQL92] conditional expression syntax.

Although, at first sight, JMS appears to be a strong contestant for a large-scale middleware, it suffers from several shortfalls: First, the entire model is centralised with respect to JMS servers. As a result, JMS servers are heavy-weight middleware components and can become bottlenecks because the JMS specification does not address the routing of JMS messages across multiple servers or the distribution of servers to achieve load-balancing. Second, content-based filtering of messages in JMS only considers the message header but not the message body. This seriously reduces the usefulness of message filtering. Finally, JMS is tightly integrated with the Java language. This has the advantage that object instances can be published in a message, but comes with the price of only supporting Java clients, which is not feasible in a large-scale, heterogeneous distributed system.

IBM WebSphere MQ

IBM WebSphere MQ [IBM02a] (formerly known as IBM MQSeries) is a message-oriented middleware platform that is part of IBM's WebSphere suite for business integration. Messages are stored in message queues that are handled by queue managers. A *queue manager* is responsible for the delivery of messages through server-to-server channels to other queue managers. A message has a header and an application body that is opaque to the middleware. No type-checking of messages is done by the middleware. Several programming language bindings of the API to send and receive messages to and from queues exist, among them a JMS interface. WebSphere MQ comes with advanced messaging features, such as transactional support, clustered queue managers for load-balancing and availability, and built-in security mechanisms.

Having many features of a request/reply middleware, WebSphere MQ is a powerful middleware, whose strength lies in the simple integration of legacy applications through loosely-coupled queues. Nevertheless, it cannot satisfy the more complex many-to-many communication needs of modern large-scale applications, as it lacks natural support for multi-hop routing and expressive subscriptions. To address these shortcomings, the Gryphon Event Broker, described in Section 2.2.2, has been added to the WebSphere suite. It extends the basic messaging framework with multi-broker, content-based publish/subscribe functionality.

2.1.3 Other Middleware

Apart from the previously discussed mainstream middleware platforms, there are also new middleware architectures that are targeted at specialised application domains. In the following we present three approaches, web services, reflective middleware, and tuple spaces, that try to face the novel middleware requirements introduced by large-scale distributed systems. Our design for an event-based middleware benefits from drawing on these ideas and techniques, as these middlewares are more deeply founded in current research.

Web Services

Web services [W3C03a, W3C03b, W3C03c] have received much interest in the recent past, as they have been endorsed by many major software companies. They are an open middleware framework for application interaction on the Internet that is based on open standards and existing web protocols, such as XML and HTTP. The web services framework is divided into three areas — communication protocols, service descriptions, and service discovery.

Simple Object Access Protocol (SOAP). SOAP is a communication protocol that supports both asynchronous messaging and synchronous RPC among web applications. It is language-independent through using XML Schema [W3C01a, W3C01b, W3C01c] to express message formats and invocation interfaces. SOAP messages are transported by existing protocols, such as HTTP or SMTP. Programming language bindings exist that tie SOAP RPC calls to client and server interfaces.

Web Services Description Language (WSDL). Interfaces of remote objects that implement a web service are described in WSDL. WSDL specifies the precise SOAP messages that need to be exchanged when invoking a web service. Language-specific types in the remote method signatures are mapped to corresponding XML Schema types.

Universal Description, Discovery, and Integration (UDDI). The final part of the web services framework is the UDDI directory that enables clients to discover WSDL descriptions of services that they want to invoke. Clients use SOAP messages to update and query the UDDI directory.

Since web services are targeted at the development of internet applications, they support synchronous and asynchronous communication. Synchronous request/reply is built on top of asynchronous messaging by pairing request and reply messages. A web service can export a synchronous and an asynchronous interface to the outside world. However, the interaction between clients and services is restricted to one-to-one communication. This prevents web services from being used for large-scale information dissemination. In other words, web services lack publish/-subscribe functionality.

A design choice of the web services framework was the use of language-independent, open web standards. This has the benefit that no new protocols or data formats need to be invented. XML Schema is a powerful and widely-adopted specification for describing and expressing data. Web services can take advantage of this and thus integrate well within a heterogeneous environment. We follow the same approach when designing an event-based middleware.

Reflective Middleware

Reflection was first introduced as a programming language concept for languages that can reason about and act upon themselves [Mae87]. The notion has been extended to middleware, such that a *reflective middleware* [KCBC02, Cou01] is capable of *inspecting* its own state and *adapting* it at runtime. This helps the middleware operate in dynamic environments, where the application requirements and the underlying network properties, such as resource availability and link connectivity, are constantly changing during the lifetime of the middleware. In such environments, applications demand that the middleware exposes some of its internal state to them and that it accepts application-specific requests, for example, to reconfigure the network protocol stack when the wireless connectivity of a client is low. Contrary to traditional middleware, which hides its state in order to give a high-level abstraction to applications, a reflective middleware exposes this state by means of *meta-interfaces*. Its architecture is usually component-based so that parts of the middleware can be replaced and reconfigured at runtime.

The *OpenORB* reflective middleware [BCA+01] is a CORBA implementation that leverages reflective techniques. The middleware divides into components that each have their own meta-level components for reflection, populating the component's meta-space. *Meta-spaces* are partitioned into distinct meta-space models, which group together meta-level components that deal, for example, with structural or behavioural reflection. Behavioural reflection is implemented with *interceptors* that can alter interface invocations. Another reflective middleware project is *dynamicTAO* [KRL+00], which is an extension of the TAO real-time ORB. It has *component configurators* that support the on-the-fly replacement of middleware components. Interceptors can also inject application code in the execution path of a remote method invocation.

Reflective features in a middleware for large-scale systems are essential to cope with changes in the environment. Therefore, any novel middleware design, such as an event-based middleware, should be component-based and harness reflective techniques. However, current reflective middleware is strongly influenced by synchronous request/reply communication as proposed by CORBA. In the more general case of many-to-many communication, it becomes even more important to involve the application in the complex interactions between middleware components.

Tuple Spaces

An abstraction for distributed shared memory computing is tuple spaces, first advocated in the *Linda* programming language [Gel85]. A *tuple space* is a shared collection of ordered data *tuples* that supports three operations: `read` to read a tuple from the tuple space that matches a template, `out` to read and remove a tuple, and `in` to insert a new tuple into the space.

The *JavaSpaces* [Sun03a] specification uses Java's type system to express elements in a tuple space and rules for matching them. In addition, it supports transactional access to the tuple space from distributed hosts. It extends the original Linda interface with a `notify` operation that makes an asynchronous callback when a matching element is inserted into the space. To do this, it follows Jini's distributed event specification, transmitting a `RemoteObject` to an

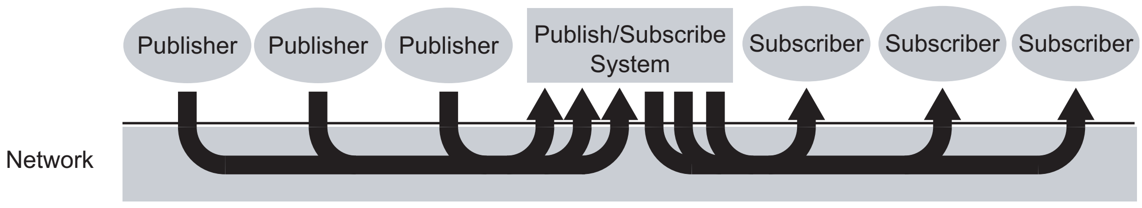


Figure 2.3: Components in a publish/subscribe system

`EventListener`. IBM's implementation of tuple spaces is called *T-Spaces* [Wyc98]. T-Spaces come with a tighter database integration and supports the dynamic addition of new operators over the tuple space.

Tuple spaces with asynchronous callbacks as provided by JavaSpaces resemble publish/subscribe systems. A `notify` operation can be seen as a subscription, whereas the addition of a new tuple is a publication. This behaviour is made explicit in the distributed asynchronous collections described in Section 2.2.2. The challenge, however, is not to rely on a centralised tuple space, but instead implement a partitioned space in a scalable and efficient way without violating the original model. Many of the problems here are the same as for an event-based middleware implementation.

2.2 Publish/Subscribe Systems

The *publish/subscribe model* [EFGK03] is an asynchronous, many-to-many communication model for distributed systems. It is an efficient way to disseminate data to a large number of clients depending on their interests. From a programmer's point of view, it is simple to use because it ensures that information is delivered to the right place at the right time. In the publish/-subscribe model, there are two different types of clients, *information producers* and *information consumers*, that disseminate data in the form of events.

Definition 2.2 (Publish/Subscribe Model) *The* Publish/Subscribe Model *has* event publishers *that produce* events *(or* event publications*) and* event subscribers *that receive them. Event subscribers describe the kind of events that they want to receive with an* event subscription. *Events coming from event publishers will subsequently be delivered to all interested event subscribers with matching interests.*

A publish/subscribe system implements the publish/subscribe model and provides a service to applications by storing and managing subscriptions and asynchronously disseminating events. An example of a publish/subscribe system with connected publishers and subscribers is given in Figure 2.3. An important feature of the publish/subscribe model is that clients are decoupled. A publisher does not need to know all subscribers that receive an event and, similarly, subscribers do not know the identity of publishers that send events to them. All communication between them is handled by the publish/subscribe system. Thus, this *loose coupling* removes dependencies between clients so that the publish/subscribe model is a scalable communication paradigm for large-scale systems.

Many publish/subscribe systems exist that implement different variants of the publish/subscribe model. A major distinction is how event subscriptions express the interest of event subscribers. Two main forms of the publish/subscribe model have resulted from this, topic-based (Section 2.2.1) and content-based (Section 2.2.2) publish/subscribe. We will use this distinction to classify current commercial and academic publish/subscribe systems in the remainder of this section. Moreover, the publish/subscribe model does not dictate whether the implementation is centralised or distributed, but a centralised implementation can restrict scalability. As a result, this survey of publish/subscribe systems focuses on distributed implementations.

2.2.1 Topic-Based Publish/Subscribe

The earliest variant of the publish/subscribe model is *topic-based publish/subscribe.* In topic-based publish/subscribe, event publishers publish events with respect to a *topic* or *subject.* Event subscribers specify their interest in a topic and receive all events published on this topic. Therefore topics can be seen as groups in group communication [Pow96]. This makes the implementation of a topic-based publish/subscribe system simple and efficient since it can be built on top of a group communication mechanism such as IP multicast [Dee89].

However, the application scenarios in Section 1.2 indicate that event subscribers benefit from a more precise specification of interest than just a topic name. Subdividing the event space into topics has the disadvantage that it is inflexible and may lead to subscribers having to filter events coming from general topics. Some topic-based publish/subscribe systems alleviate this effect with hierarchical topics that help structure the topic space. Wildcards in topics names are often supported to allow subscription to several topics simultaneously. For example, a subscription to `/ActiveOfficeEvent/PeopleEvent/*` will deliver all events in subtopics of the `PeopleEvent` topic.

Most commercial publish/subscribe implementations are topic-based, such as Java JMS providers (described in Section 2.1.2) or the TIBCO Rendezvous messaging system [TIB99]. Next, we will look at the Information Bus architecture, an early topic-based publish/subscribe system, but after this we will concentrate on content-based publish/subscribe, which is more suitable for an event-based middleware.

Information Bus

A distributed implementation of a topic-based publish/subscribe system is the *Information Bus* [OPSS93]. *Data objects* (events) are published on a logical information bus that links publishers with subscribers. The bus is implemented using Ethernet broadcasts in a LAN. Events carry a hierarchical subject string that subscribers specify in subscriptions, potentially including wildcards. Reliable event delivery is achieved by UDP packets with a simple retransmission protocol. Systems spanning multiple networks can be supported by the architecture with *information routers* that bridge Ethernet broadcasts across networks depending on subscriptions.

Apart from the obvious lack of expressiveness in topic-based publish/subscribe, the Information Bus architecture suffers from limited scalability as the filtering of events is left to subscribers. The network can easily be overwhelmed by too many event broadcasts. With a moderate number of events, this architecture is a simple and effective implementation of a topic-based publish/subscribe system in a LAN, but otherwise not applicable to large-scale systems.

2.2.2 Content-Based Publish/Subscribe

In a content-based publish/subscribe system, the structure of an event subscription is not restricted — it can be any function over the content of an event. This introduces a trade-off

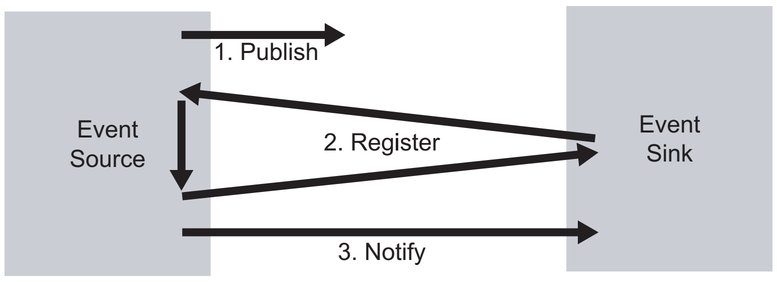


Figure 2.4: The publish-register-notify paradigm in the CEA

between scalability and expressiveness [CRW99]. The more expressive an event subscription becomes, the more difficult it is to evaluate it, increasing the overhead in the publish/subscribe system. Many schemes for expressing content-based event subscriptions have been proposed. A content-based subscription usually depends on the structure of the event. As seen before, the structure of an event can be binary data, name/value pairs, semi-structured data such as XML, or even programming language classes with executable code. A subscription is often expressed in a *subscription language* that specifies a filter expression over events.

The following is an overview of research efforts that led to content-based publish/subscribe systems. When describing these projects, we evaluate how suitable they are for the construction of large-scale distributed systems with respect to scalability of event dissemination and with emphasis on any middleware functionality.

CEA

The *Cambridge Event Architecture* (CEA) [BBHM95, BMB$^+$00] was created in the early 90s to address the emerging need for asynchronous communication in multimedia and sensor-rich applications. It introduced the *publish-register-notify* paradigm for building distributed applications. This design paradigm allows the simple extension of synchronous request/reply middleware, such as CORBA, with asynchronous publish/subscribe communication. Middleware clients that become *event sources* (publishers) or *event sinks* (subscribers) are standard middleware objects.

The interaction between an event source and sink is illustrated in Figure 2.4. First, an event source has to advertise (*publish*) the events that it produces; for example, in a name service. In addition to regular methods in its synchronous interface, an event source has a special `register` method so that event sinks can subscribe (*register*) to events produced by this source. Finally, the event source performs an asynchronous callback to the event sink's `notify` method (*notify*) according to a previous subscription. Note that event filtering happens at the event sources, thus reducing communication overhead. The drawback of this is that the implementation of an event source becomes more complex since it has to handle event filtering.

Despite the low latency, direct communication between event sources and sinks causes a tight coupling between clients. To address this, the CEA includes *event mediators*, which can decouple event sources from sinks by implementing both the source and sink interfaces, acting as a buffer between them. Chaining of event mediators is supported but general content-based routing, as done by other distributed publish/subscribe systems, is not part of the architecture. More recent work [Hom02] investigates the federation of separate CEA event domains using contracts that are enforced by special mediators acting as gateways between domains. A Java implementation of the CEA, *Herald* [Spi00], supports storage of events.

The design goal of the CEA is to seamlessly integrate publish/subscribe with standard middleware technology. Therefore, events are strongly-typed objects of a particular event class and are statically type-checked at compile time. Initially, subscriptions were template-based for equality matching only, but they were then extended with a predicate-based language with key/value pairs. These subscriptions are type-checked dynamically at runtime. Furthermore, the CEA provides a service for complex subscriptions based on composite event patterns [Hay96]. This is an important requirement for an event-based middleware, and we present our approach for detecting composite events in Chapter 7.

COBEA. The CEA was implemented on top of CORBA in the *CORBA-Based Event Architecture* (COBEA) [MB98]. COBEA can be regarded as a precursor to the CORBA Event Service that was described in Section 2.1.1. Events are passed between event sources and sinks as parameters in CORBA method calls because fully-fledged CORBA objects would be too heavy-weight as events. Event clients can by typed or untyped — a typed client encodes the structure of an event type in an IDL `struct` datatype, whereas an untyped client uses the generic `any` datatype. Type-checking for typed clients is done by the IDL compiler. The subscription language consists of a conjunction of predicates over the attributes defined in the event type.

ODL-COBEA. The use of CORBA IDL to express event types is cumbersome since its original purpose is the specification of interfaces for remote method calls. In [Pie00], COBEA is extended with an event type compiler that transforms event type definitions in the *Object Definition Language* (ODL) [CBB+97] into appropriate CORBA IDL interfaces. The ODL language is a schema language defined by the Object Data Management Group (ODMG). With ODL, objects can be described language-independently for storage in an object-oriented database. The advantage of using ODL for event definitions is that it provides support for persistent events because it unifies the mechanisms for transmission and storage of events [BHM+00].

An example of an ODL-defined event type, as it would be used in the Active Office application scenario, is given in Figure 2.5. Event types consist of a set of typed attributes and form an ODL inheritance hierarchy, in which all types are derived from the `BaseEvent` ancestor class. The `BaseEvent` type has attributes that all event types inherit, namely a unique `id` field, a `priority` field, a `source` field with the name of the event source that generated this event, and a `timestamp`. ODL-COBEA is aware of inheritance relationships between event types and supports *supertype subscriptions*. When an event subscriber subscribes to an event type, it will also receive any published events that are of a subtype of the type specified in the subscription. This means that an event subscriber that subscribes to the `BaseEvent` type will consequently receive all events published at a given event source.

The CEA and in particular the ODL-COBEA implementation recognise the importance of type-checking for events in a publish/subscribe system. The object-oriented approach for defining event types cleanly integrates with current object-oriented programming languages and middle-

```
class LocationEvent extends BaseEvent {
  attribute short id;
  attribute string location;
  attribute long lastSighting;
};
```

Figure 2.5: An ODL definition of an event types in ODL-COBEA

ware architectures. Static type-checking, as done by an event type compiler, does not introduce a runtime cost, but it tightly couples event sinks to sources.

The main disadvantage of the CEA is the lack of content-based event routing between event mediators. This limits the scalability of the architecture as it forces a subscriber to know the publisher (or mediator) that offers a particular event type. In addition, it makes the implementation of event sources challenging because they are required to perform event filtering depending on subscriptions. Several distributed content-based publish/subscribe systems were proposed after the CEA to address these problems.

Siena

One of the first implementations of a distributed content-based publish/subscribe system is the *Scalable Internet Event Notification Architecture* (SIENA) [Car98, CRW01]. SIENA is a multi-broker event notification service that is targeted at Internet-scale deployment. Event publishers and subscribers are clients that first need to connect to an *event broker* in the logical overlay network. Events published by publishers are then routed through the overlay network of brokers depending on the subscriptions submitted by subscribers. This process of *content-based routing* [CRW00] can be viewed as a distributed implementation of an event matching algorithm so that events are delivered to all interested subscribers.

A SIENA event, called a *notification*, consists of a set of typed attributes. Subscriptions are conjunctions of *event filters*, which are predicates over the event attributes. A subscription *covers* a notification if and only if all event filters in the subscription hold when evaluated with respect to the notification. A subscription s_1 covers another subscription s_2 (or an advertisement a_1) if and only if any notification that is covered by s_2 (or a_1) is also covered by s_1. A more detailed presentation of the covering relation will be be given in Section 3.3.1.

Although several topologies for the network of event brokers in SIENA are proposed, we will only consider the *general peer-to-peer topology* because it is most realistic for a large-scale network. The content-based routing algorithm for this topology uses three types of messages, *advertisements*, *subscriptions*, and *notifications*. It is based on the idea of *reverse path forwarding* of messages in broadcast algorithms [DM78]. A formal description of the SIENA routing algorithm can be found in [Müh02].

Advertisements. An advertisement message is sent by publishers to express their intention to publish events. It sets up routing state in the broker network so that future notifications can be delivered to all subscribers. The advertisement must cover all notifications that the publisher intends to publish. An advertisement message is broadcast through the entire overlay broker network unless it is covered by a previous advertisement.

Subscriptions. A subscription message contains the specification of interest submitted by a subscriber. It creates state in the overlay broker network by following the reverse path of any advertisement that covers this subscription.

When advertisement/subscription messages propagate through the network, they create entries in advertisement/subscription routing tables that store (1) the advertisement/subscription message, (2) the last hop, and (3) the next hop for this message.

Notifications. Notification messages carry the events that are delivered to subscribers. Notifications follow the reverse path of any subscriptions that cover them. No state in event brokers is created for notifications.

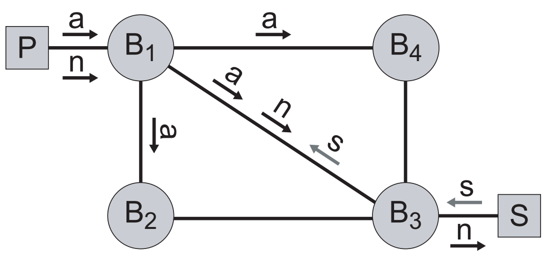


Figure 2.6: Content-based routing by event brokers in Siena

The operation of the content-based routing algorithm together with the state kept in routing tables at event brokers is illustrated in Figure 2.6. In this example, event publisher P sends an advertisement a that is stored at event brokers B_1, B_2, B_3, and B_4. After that, event subscriber S creates a subscription s that is covered by the previous advertisement a. Therefore, s follows the reverse path of a through the event broker B_3 and B_1, and state in subscription routing tables is set up. Finally, P publishes the notification n that follows the reverse path of s and reaches subscriber S.

A characteristic feature of this routing algorithm is that subscriptions are pushed close to the publishers. This allows filtering to take place immediately after the publication of a notification, thus saving network bandwidth as only events that have interested subscribers are propagated further through the network. The propagation of a new, more specific advertisement/subscription can be avoided if it is covered by a previous, more general advertisement/subscription. The rationale behind this is that the new advertisement/subscription would not set up a new path through the overlay network and hence can be discarded. This helps reduce the state kept in routing tables and the bandwidth used in the network for advertisement/subscription messages.

Siena is a promising approach for a large-scale middleware but it lacks support for type-checking of events. The complete freedom given to publishers to advertise and publish any event makes it harder to catch type-mismatch errors during system development. Even though the idea of *event patterns* is introduced as a higher-level service, little detail is given on detection and temporal issues, as addressed in Chapter 7.

Considering the content-based routing algorithm, the scalability of Siena suffers from the fact that, in the worst case, advertisement messages have to reach every event broker in the overlay broker network. This introduces an unknown delay until a message has been successfully processed by all the event brokers and reduces the robustness of the routing algorithm. In addition, the topology of the overlay network of event brokers is static and must be specified at deployment time. The efficiency of the content-based routing will therefore depend on the quality of the overlay network topology. Having a static topology is not feasible for a large-scale system, that may involve thousands of event brokers running at geographically dispersed sites.

Gryphon

The *Gryphon* project at IBM Research [IBM01] led to the development of an industrial-strength, reliable, content-based event broker that is now part of IBM's WebSphere suite as the IBM WebSphere MQ Event Broker [IBM02b]. It is a mature publish/subscribe middleware implementation with a JMS interface that provides a redundant, topic- and content-based multi-broker publish/subscribe service. The Gryphon event broker has been successfully deployed for large-

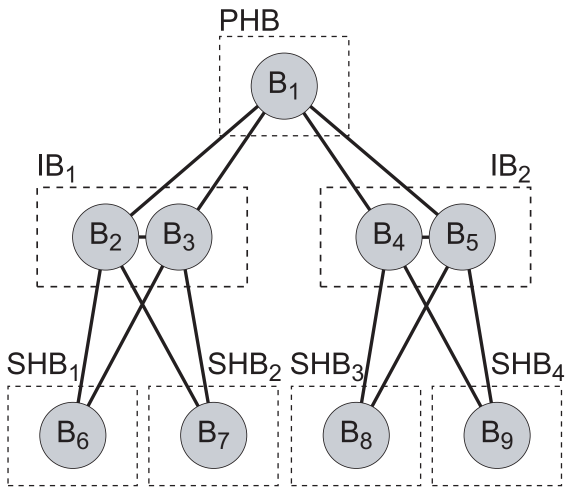


Figure 2.7: A Gryphon network with virtual event brokers

scale information dissemination at global sports events, such as the Olympic Games. It includes an efficient event matching engine, a scalable routing algorithm, and security features.

Gryphon is based on an information flow model for messaging [BKS+99]. An *information flow graph* (IFG) specifies the exchange of information between information producers and consumers. Information flows can be altered by (1) filtering, (2) stateless transformations, and (3) stateful transformations (aggregation). A logical IFG is mapped onto a physical event broker topology. Figure 2.7 shows an example of a Gryphon deployment. Nodes in the IFG are partitioned into a collection of *virtual brokers* PHB, $IB_{1,2}$, and SHB_{1-4}, which are then mapped onto clusters of physical event brokers called *cells*. Similarly, edges connecting nodes in the IFG are *virtual links* that map onto *link-bundles*, containing multiple redundant connections between event brokers for reliability and load-balancing.

An event broker that has publishing clients connected to it is called a *publisher-hosting broker* (PHB). It contains *publisher endpoints* (or *pubends*), which represent a collection of publishers that enter information into the IFG. Correspondingly, a *subscriber-hosting broker* (SHB) consumes information through one or more *subscriber endpoints* (or *subends*) from the IFG according to its subscriptions. An event broker that is neither publisher-hosting nor subscriber-hosting is an *intermediate broker* (IB). The topology mapping is statically defined at deployment time, although more recent work includes dynamic topology changes due to failure and evolution. Several extensions are implemented as part of the Gryphon event broker.

Guaranteed Delivery. A *guaranteed delivery service* [BSB+02] provides exactly-once delivery of events, as required for JMS persistent events. The propagation of information (*knowledge*) from pubends to subends is modelled with a *knowledge graph*. Lost knowledge due to message loss causes *curiosity* to propagate up the knowledge graph and trigger the re-transmission of events. Curiosity is implemented as *negative acknowledgement* (NACK) messages sent by subscriber-hosting brokers. A subscriber that remains connected to the system is guaranteed to receive a gapless ordered filtered subsequence of the *event stream* published at a pubend. A more detailed description of guaranteed delivery and how it can be extended to address congestion in an event-based middleware will be given in Chapter 6.

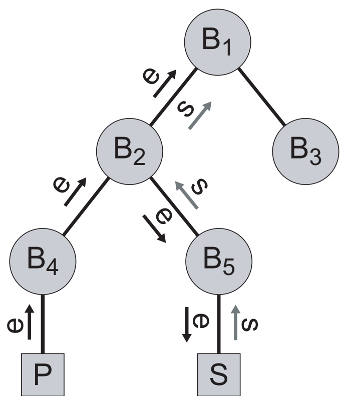


Figure 2.8: Hierarchical event routing in JEDI

Durable Subscriptions. The *durable subscription service* [BZA03] guarantees exactly-once delivery despite periods of disconnection of event subscribers from the system. This means that the event stream is buffered while a subscriber is not available and replayed upon re-connection. As for the guaranteed delivery service, an event log is kept at publisher-hosting brokers and cached at intermediate brokers.

Relational Subscriptions. The final extension is the *relational subscription service* [JS03]. The goal here is to implement the stateful transformations supported by Gryphon's IFG model, combining messaging with a relational data model. Relational subscriptions can be seen as a continuous query over event streams, providing event subscribers with the expressiveness of a relational language. This relates to the requirement for composite event detection in an event-based middleware, which will be discussed in Chapter 7.

The Gryphon event broker includes many of the features that a distributed systems programmer expects from an event-based middleware. However, the overlay network of event brokers is static, as it is defined in configuration files at deployment time. This makes it difficult for the middleware to adapt to changed network conditions. Failure within a cell of event brokers can be tolerated, but major changes to the IFG cannot be compensated for. Although composite event detection is provided by relational subscriptions, a relational data model for messaging can prove to be too heavy-weight for many applications.

JEDI

The *Java Event-Based Distributed Infrastructure* (JEDI) [CNF01] is a Java-based implementation of a distributed content-based publish/subscribe system from the Politecnico di Milano, Italy. Events in JEDI have a name and a list of values for event parameters. Subscriptions are specified in a simple pattern matching language. A JEDI system consists of *active objects*, which publish or subscribe to events, and *event dispatchers*, which route events. Event dispatchers are organised in a tree structure, and routing is performed according to a *hierarchical subscription strategy*. Subscriptions propagate upwards in the tree and state about them is maintained at the event dispatchers. Events also propagate upwards but follow downward branches whenever they encounter a matching subscription, as shown in Figure 2.8. Advertisements are not used to restrict the propagation of subscriptions.

The system has been extended to support mobile computing [CN01]. Event dispatchers support `moveOut` and `moveIn` operations that enable subscribers to disconnect and reconnect at a different point in the network. There is no single event dissemination tree for all subscriptions, but instead a tree is built dynamically as a *core-based tree* [BFC93]. The core, called a *group leader*, has to make a global broadcast to announce its presence. A new event dispatcher, wanting to become part of the dissemination tree, directly contacts the group leader. The group leader then delegates the request to an appropriate event dispatcher in the dissemination tree, which becomes the parent of the new node. As a downside, this algorithm requires that every event dispatcher must have knowledge of all group leaders in the system.

An approach for dynamically reconfiguring the dissemination tree is proposed in [PCM03]. In the *strawman strategy*, the failure of a link between event dispatchers triggers the propagation of unsubscriptions that ensure that the routing tables stay consistent. In the same fashion, new links in the overlay network cause the flow of subscriptions. To prevent unsubscriptions from removing more state from routing tables than necessary, unsubscriptions are delayed by a timeout value, although finding a suitable value is difficult in practice.

Although the JEDI system addresses the need for dynamic event dissemination trees, the overlay network topology between event dispatchers is still tree-based. This makes it challenging to achieve robustness in the system, as a single link failure between two event dispatchers partitions the tree and can cause event loss. Moreover, the scalability of the system suffers from the fact that subscriptions are broadcast to all event dispatchers to establish global knowledge about group leaders.

Elvin

Elvin [SA97] is a notification service for application integration and distributed systems monitoring developed by the Distributed Systems Technology Centre in Australia. It features a security framework, internationalisation, and pluggable transport protocols, and has been extended to provide content-based routing of events [SAB$^+$00]. Events are attribute/value pairs with a predicate-based subscription language. An interesting feature of Elvin is a *source quenching mechanism*, where event publishers can request information from event brokers about the subscribers currently interested in their events. This enables publishers to stop publishing events when there are no subscriptions, reducing computation and communication overheads.

Clients for a wide range of programming languages are available, which led to the implementation of many notification applications. Applications, such as a tickertape, were evaluated as means for collaboration in a pervasive office environment [FKM$^+$02]. More recent work investigates event correlation and support for disconnected operation in mobile applications [SAS01].

Rebeca

The REBECA [FMB01] project investigates the potential of publish/subscribe technology for large-scale e-commerce applications. The focus lies on scalable routing algorithms and software engineering techniques for the design and implementation of event-based business applications. In [Müh02, MFGB02], current content-based routing algorithms are formalised and evaluated in the REBECA infrastructure. It is shown that advanced routing algorithms with covering and advertisements have a substantial advantage over naïve approaches that rely on flooding the network with events. In addition, the idea of *subscription merging* is introduced [MFB02] as a way of reducing the state kept in routing tables at event brokers.

Returns	API Call	Parameters
`Object`	*get*	`()`
`boolean`	*add*	`(Object o)`
`boolean`	*contains*	`(Notifiable n, Condition c)`

Table 2.2: The interface of a distributed asynchronous collection

Event-based systems benefit from a modular design with novel structuring concepts, such as *event scopes* [FMMB02, FMG03]. An event scope restricts the visibility of published events to a subset of subscribers in the system. Scopes can be nested and are able to re-publish events. When crossing scope boundaries, an event may be transformed according to an *event mapping*, similar to gateways in event federations. Therefore, event scopes address the heterogeneity of modern applications and are a powerful mechanism for structuring large-scale applications.

Narada Brokering

The *Narada Brokering* project [PF03] aims to provide a unified messaging environment for grid computing, which integrates grid services, JMS, and JXTA. It is JMS compliant (see Section 2.1.2), but also supports a distributed network of brokers as opposed to the centralised client/server solution advocated by JMS. The JXTA specification [Gon02] is used for peer-to-peer interactions between clients and brokers.

Events can be XML messages that are matched against XPath [W3C99b] subscriptions by an XML matching engine. The network of brokers is hierarchical, built recursively out of clusters of brokers. Every broker has complete knowledge of the topology, so that events can be routed on shortest paths following the broker hierarchy. In general, there is the additional overhead of keeping event brokers organised hierarchically, which can be costly. Dynamic changes of the topology are propagated to all affected brokers.

Type-Based Publish/Subscribe

The work on *type-based publish/subscribe* [Eug01] recognises the need for better integration of publish/subscribe communication with mainstream, object-oriented programming languages. In type-based publish/subscribe, events are first-class programming language objects with fields and methods. Subscriptions specify the programming language type of objects that a client is interested in, while observing the subtyping relation. For a more fine-grained filter specification, conditions using the methods in the event object can be provided by subscribers relying on Java's structural reflection features. An even tighter integration is proposed by adding new primitives to the Java language for publishing and subscribing to events [EGD01]. Note that our definition of type-based routing in Section 4.3.3 differs from the one presented here.

In a prototype, type-based publish/subscribe is used to implement *distributed asynchronous collections* (DACs) [EGS00] in Java. A DAC is an asynchronous, distributed data structure that holds a collection of objects. Using Java language extensions for parametric polymorphism [BOSW98], a DAC can be parameterised to only hold Java objects of a specific type. An excerpt from its interface is shown in Table 2.2. Apart from the standard synchronous methods found in collections to add and remove objects, it has an asynchronous `contains` method that will perform a callback to the `Notifiable` interface when an object is added to the DAC that matches a `Condition`. This can be considered a subscription, whereas inserting an object into a DAC corresponds to making a publication. DACs are implemented efficiently using probabilistic, gossip-based multicast algorithms [EGH+03].

Advantages of type-based publish/subscribe are that the object-oriented principle of encapsulation is not violated because fields in an object are only accessed via its methods, and no separate subscription language is necessary to express interest in events. As a consequence, publish/subscribe integrates seamlessly with the programming language, which is important for an event-based middleware. The penalty to be paid is that event matching is less efficient as it amounts to executing method calls that may involve reflection. When considering the semantics of the DAC interface, some operations are not intuitive since, unlike standard tuple spaces, they may or may not remove an object from the DAC after a notification.

2.3 Peer-to-Peer Systems

The term *peer-to-peer* [Ora01] first emerged in the context of file-sharing applications on the Internet. It can be regarded as a way of building distributed applications that contrasts with the traditional approach of a client/server architecture. Most peer-to-peer systems deal with the sharing of a resource, such as storage or computation, and are often implemented as *overlay networks* [Eri94] at the application-level. Research into peer-to-peer systems has resulted in a diverse range of algorithms and applications, which makes it difficult to give a consistent definition of what a peer-to-peer system is. The following is an attempt at highlighting the main characteristics that make a distributed system a peer-to-peer system.

Definition 2.3 (Peer-to-Peer System) *A* Peer-to-Peer System *is a decentralised, distributed system that consists of symmetric nodes called* peers*. It is self-organising and capable of adapting to changes such as failure.*

A primary feature of peer-to-peer systems is the unstable connectivity between peers. This means that failure is common in a peer-to-peer system, with nodes joining and leaving at all times. Robustness has to be built into all mechanisms so that correct operation of the system is ensured even under failure of a substantial number of peers. The fully decentralised approach of peer-to-peer systems results in highly scalable systems that operate on an Internet-scale, making peer-to-peer techniques attractive for large-scale middleware with the complexity of many components.

The first generation of peer-to-peer systems was unstructured, and thus suffered from scalability problems caused by flooding the system with messages [CP02]. The second generation had a stronger foundation in research and managed to deliver the promised advantages of peer-to-peer technology. In the next section we describe a common form of peer-to-peer overlay routing layer that provides the abstraction of a distributed hash table. This is a powerful distributed data structure that will be used for the implementation of our event-based middleware. Peer-to-peer techniques are also applied to application-level multicast systems, as will be discussed in Section 2.3.2.

2.3.1 Distributed Hash Tables

A *distributed hash table* (DHT) [Cla99] is a scalable data structure for building large-scale distributed applications. It maps a *key* to a distributed storage location at a particular node in the network. Instead of having global knowledge, nodes only need to know about a small subset of all existing nodes. Requests for a key are routed via the overlay network to the destination node that the key hashes to, even when nodes are constantly joining and leaving the DHT. The load of storing data in the hash table is therefore spread across all nodes in the system. The routing algorithm for the DHT builds a *small-world network* [WS98], which has a small diameter, but is highly clustered, so that every node can be reached in a logarithmic number of hops. In the following we will describe four implementations of DHTs with peer-to-peer routing algorithms that can facilitate the content-based routing of events.

Pastry

Pastry [RD01] developed at Microsoft Research in Cambridge is a peer-to-peer location and routing substrate with locality properties that forms a self-organising, resilient overlay network that can scale to millions of nodes. Its main operation is a `route(message, key)` function that reliably routes a `message` to the Pastry node that is responsible for storing the `key`. Messages take $O(\log N)$ hops on average where N is the number of nodes in the Pastry network. The overlay network of nodes is organised so that routes with a lower proximity metric, such as latency or bandwidth, are preferred. In addition, two routes to the same destination converge quickly [CDHR02]. Several applications are built on top of Pastry, such as PAST [RD01], a persistent global store, and Scribe [RKCD01], an application-level multicast system, which will be described in Section 2.3.2.

The routing algorithm of Pastry relies on the fact that each Pastry node has a unique node identifier, called a *nodeID*. NodeIDs populate a 128-bit namespace that is uniformly distributed and are grouped into digits with base 2^b for a given value of b. The functionality of a DHT is implemented by routing a message to a live node with a nodeID that is numerically closest to the hash key. The routing of messages relies on two data structures, a routing table and a leaf set, maintained by each node.

Routing Table. The routing table has $\log_{2^b} N$ rows with $2^b - 1$ columns. The rows contain entries for nodes whose nodeID matches the current nodeID in the first d digits but then differs afterwards. Among several candidate nodeIDs for an entry in the routing table, the one with the minimum proximity metric is chosen. Secondary entries are kept as backup in case the primary node fails.

Leaf Set. The leaf set has l nodeIDs as entries, which are the $l/2$ closest, numerically larger and smaller nodeIDs with respect to the current nodeID. This invariant must be maintained at all times and routing will fail if more than $l/2$ nodes with consecutive nodeIDs fail. The leaf set is also used for data replication.

Routing in Pastry is a generalisation of prefix routing. A message is forwarded to a node that shares a longer prefix with the destination nodeID than the current node. If such a node does not exist in the routing table, the message is sent to a node with a nodeID that is numerically closer to the destination. If the destination nodeID falls within the range of the leaf set, the message is sent directly to the numerically closest nodeID. The process of routing a message from node 123 to the key 333 with $b = 2$ is shown in Figure 2.9. The message is first forwarded

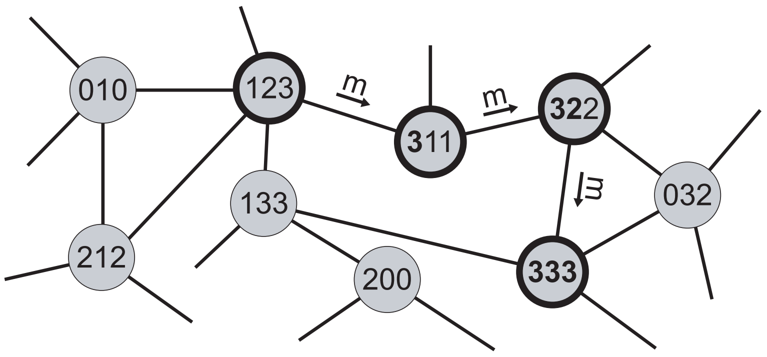


Figure 2.9: Routing a message in a Pastry network

to node 311, which is obtained from the routing table at node 123. Each hop moves the message closer to the destination node.

New Pastry nodes can join the system efficiently at runtime by following a *join protocol*. A new Pastry node first chooses a random nodeID X and then asks an arbitrary existing node A to route a join message with the key X. While the message traverses the network to a destination node Z that currently manages key X, the message acquires the consecutive rows from the routing tables of the nodes along the path to node Z. Finally, the node Z sends the routing table data and its own leaf set to the new node A, which then initialises its data structures and notifies its leaf set neighbours of its existence.

The Pastry implementation of a DHT is an efficient way of maintaining an application-level overlay network for routing. Since routing tables are optimised to reflect the physical topology of the network, the penalty of using overlay routing lies within a small constant factor [CDHR02]. A distributed event-based middleware needs to manage an overlay network of event brokers that is scalable, fault-tolerant, and efficient. Applying peer-to-peer techniques to achieve this is a promising novel approach.

Tapestry

Another implementation of a DHT routing layer that came out of the Oceanstore project at the University of California at Berkeley is *Tapestry* [ZKJ01]. Its routing algorithm is similar to Pastry's but differs in how the last routing step to a destination node is handled. Tapestry does not have leaf sets but instead uses *surrogate routing* [HKRZ02]. If the routing cannot continue because the routing table at a node does not contain an entry that matches the destination key's nth digit in row n, the message is forwarded to a surrogate node with the next higher digit at position n. The routing terminates when there is no other entry at the same level in the routing table. This process ensures that a message routed to a key with a non-existing nodeID will nevertheless map to a unique live node in the Tapestry network.

Chord

In *Chord* [SMK$^+$01], nodeIDs are organised in a circular space. Messages are routed clockwise in this space until they reach a node with a nodeID that is equal to or follows the desired destination key. Every node in the ring knows its predecessor and successor. To make routing more efficient, every node keeps a *finger table* with shortcuts. The ith entry in the finger table contains the identity of the first node whose nodeID succeeds the current nodeID by at least 2^{i-1} in the circular space. By using the finger table to route a message as close to the destination key as possible, the average hop count will be logarithmic.

CAN

The *content-addressable network* (CAN) [RFH+01] differs from the previous DHT schemes since nodeIDs lie on an d-dimensional torus. The logical space is divided into hyper-rectangular *zones* with a unique CAN node responsible for storing all keys that fall within a given zone. A CAN node maintains a routing table with the identities of all its neighbours in the logical space. Messages are routed on the straight line path through the space until they reach the destination zone. The average hop count for routing a message is $O(N^{1/d})$, which is slightly less efficient compared to Pastry or Tapestry. A new node is added to the network by dividing an existing zone in half, splitting the key space between the old and new node. To preserve efficiency, the logical space must be partitioned into equally-sized zones. An application-level multicast scheme over CAN is presented in the next section.

2.3.2 Application-Level Multicast

Since native IP multicast support is not ubiquitously deployed on the Internet, multicast functionality for information dissemination is often implemented over unicast links at the application-level. In this section *application-level multicast* schemes [CRZ00] built over an overlay network of nodes are introduced. The general strategy is to construct an information dissemination tree that has the multicast group members as leaf nodes. Several approaches were proposed to build application-level multicast on top of a distributed hash table offered by peer-to-peer routing substrates, thus taking advantage of self-organisation and locality properties. The resulting application-level multicast service is equivalent to topic-based publish/subscribe. Hermes, our event-based middleware, extends simple application-level multicast routing to provide a content-based routing algorithm with filtering (see Chapter 4).

Scribe

Scribe [RKCD01] is a large-scale event notification infrastructure built on top of Pastry. Scribe uses Pastry to manage multicast groups associated with a topic. Each topic forms a multicast tree depending on the subscribers interested in the topic. Scribe takes advantage of Pastry's locality properties so that the multicast tree is optimised with respect to a proximity metric. The overlay multicast scheme implemented is similar to core-based trees [BFC93] since there is a single well-known node that acts as the root of the multicast tree. Pastry's randomisation properties ensure that the multicast tree remains well-balanced. Scribe supports three main operations, which are (1) *creating* a topic, (2) *subscribing* to a topic, and (3) *publishing* a message on a topic.

Any Scribe node can create a new topic by routing a message via Pastry to the node that is numerically closest to the hash of the topic name. This node then becomes the *rendezvous point* for the topic and functions as the root of the multicast tree. A subscriber is added to a topic by sending a subscription message towards the rendezvous point. Along the path of the message, each node becomes a member of the multicast tree called a *forwarder* and adds the previous hop to its *children table*. The propagation of the subscription message terminates when

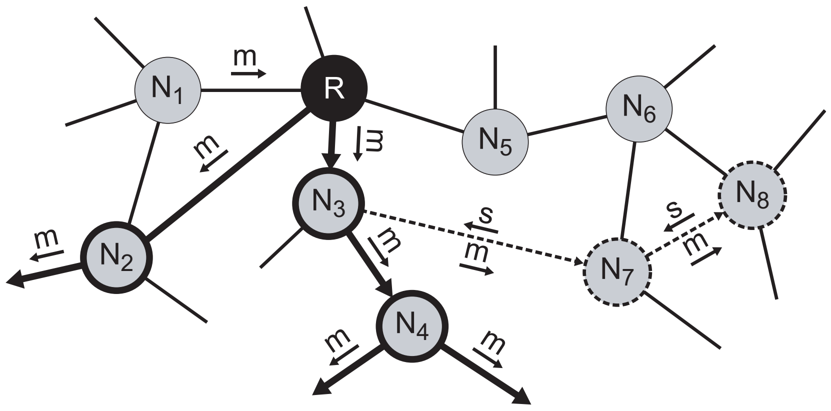


Figure 2.10: Addition of a subscribing node in Scribe

a node is reached that is already a forwarder in the multicast tree. The addition of node N_8 as a subscriber with the rendezvous point at node R is illustrated in Figure 2.10. Any node can publish a message by sending it to the rendezvous point for the topic, which will in turn propagate it along the multicast tree to all subscribed nodes, as demonstrated by node N_1.

The Pastry overlay is also used to repair the multicast tree when nodes fail. When a forwarder discovers the failure of its parent, it resends the subscription message to the rendezvous point. Pastry's routing algorithm ensures that the message takes a different path to the rendezvous point unless a significant number of nodes in the overlay network have failed. Old state in forwarders expires due to a *soft state* approach, which discards entries in children tables that are not refreshed periodically by heartbeat messages. To prevent rendezvous points from being single points of failure, they are replicated across the nodes in the original rendezvous point's leaf set.

Although subscription messages do not need to reach the rendezvous point in the multicast tree, the same is not true for publications. This means that the rendezvous point may get overloaded when there are many publishers in the system. As a general criticism, the lack of content-based filtering of publications limits the expressiveness of subscriptions and makes Scribe unsuitable as a general-purpose event-based middleware. Nevertheless, its scalable, fault-adapting message dissemination algorithm is a good building block for an event dissemination algorithm in large-scale middleware.

Bayeux

An application-level multicast solution over Tapestry is *Bayeux* [ZZJ+01]. Its algorithm is similar to Scribe's in the sense that it uses an overlay routing layer to build a tree for a multicast topic. However, it differs in how the multicast tree is constructed. In Bayeux a subscription message is routed all the way to the root of the multicast tree. The root node updates its membership list for the topic and replies with a *tree message* destined for the new subscriber. The tree message then creates state at the forwarding nodes along its path and ensures that the new subscriber becomes part of the multicast tree.

Bayeux's approach is less scalable because the root node has to keep membership information about all nodes belonging to a multicast group. In addition, group membership management is more costly as subscription messages are routed to the root node and trigger tree messages.

To alleviate the problem of the root node becoming a bottleneck, a scheme for partitioning the multicast tree is suggested so that the load can be shared among several roots.

Multicast over CAN

A multicast mechanism over CAN is presented in [RHKS01]. The strategy adopted is different from the previous approaches since no tree is constructed in the overlay network. Instead, a separate content-addressable network within the global CAN is formed that only contains the multicast group members. To publish a message to the group, the multicast CAN is flooded with the publication so that all group members receive it. The flooding algorithm takes advantage of the structure of the n-dimensional coordinate space of CAN nodeIDs to minimise the delivery of duplicate messages. Any node in the multicast CAN can be the origin of a publication without an indirection via a root node. A comparison has shown that this scheme is less efficient than tree-based multicast approaches [CJK+03].

Overcast

Several application-level multicast schemes, such as *Overcast* [JGJ+00], exist that do not rely on a distributed hash table. Instead, a scalable and reliable single-source multicast service is provided by directly building a multicast tree out of Overcast nodes that are distributed in the network. A new node joins the multicast tree by first contacting the root node. It recursively obtains a list of children and then computes a proximity metric in order to find an optimal location for itself in the tree. This adds a higher overhead to joining a multicast group compared to DHT-based multicast.

2.4 Summary

This chapter has outlined the research that is relevant to the design and implementation of an event-based middleware, for it to become the next generation middleware for large-scale systems. We began with an overview of current middleware technology, focusing on traditional request/reply middleware and the more scalable asynchronous messaging middleware, which is increasingly replacing synchronous middleware in Internet-wide systems. Even though synchronous middleware is not suitable for a large-scale context, much can be learnt from it in terms of supported services and programming language integration.

A survey of publish/subscribe systems was given that showed the power of publish/subscribe interaction for distributed computing. However, the lack of middleware features in most publish/-subscribe systems motivates the need for an event-based middleware. In addition, many current publish/subscribe systems do not achieve the required scalability, which is why peer-to-peer techniques in the form of distributed hash tables were discussed. Distributed hash tables are scalable data structures that can be used as building blocks for information dissemination systems, such as topic-based publish/subscribe.

In the next chapter, we investigate the notion of an event-based middleware in more detail with the ultimate goal of getting an understanding of the design space for such middleware. The middleware techniques introduced in this chapter will be used as guidelines when design decisions have to be made that affect the features of an event-based middleware.

3 Event-Based Middleware

In this chapter we present our notion of an event-based middleware in more detail. This is a new kind of middleware that is targeted at the development of large-scale distributed systems and therefore substantially different from previous middleware designs. Its communication strategy follows the publish/subscribe model, and it also supports common middleware functionality. When designing a novel type of middleware, many design decisions need to be made. After the overview of relevant work in the areas of middleware, publish/subscribe, and peer-to-peer research in the previous chapter, we are now in a position to survey the design space of an event-based middleware. The aim is to develop a better understanding of an event-based middleware and its functionality, with the ultimate goal of deriving a concrete architecture, Hermes, that will be presented in the next chapter. All design decisions are motivated by the two application scenarios introduced in Section 1.2. From these, we develop a list of requirements that need to be satisfied by an event-based middleware.

We start with an overview of the concept of an event-based middleware, outlining its main features. The remainder of the chapter is then concerned with partitioning the design space. This process is guided by six requirements for an event-based middleware that are described in Section 3.2. The design of the middleware is presented in relation to five design models that are introduced and discussed in Section 3.3. By referring back to the requirements, we will make sensible design decisions in this space to refine the models for an event-based middleware into an actual architecture.

3.1 Overview

An *event-based middleware* [Pie02] can be contrasted with a simple publish/subscribe system, such as the ones that were described in Section 2.2. A publish/subscribe system is a communication infrastructure that provides a service to clients. It exports an interface to its functionality, but otherwise it is independent of its clients. In particular, a publish/subscribe system is only concerned with communication and not with any other necessary support in a distributed computing environment. The same is not true for an event-based middleware. It extends the notion of a publish/subscribe system by providing general-purpose middleware support to clients. It can be seen as the "glue" between the components of a large-scale distributed application, helping them cope with the complexity and heterogeneity of the environment. An event-based middleware is influenced by the distributed computing requirements of its clients. This means that, for example, it tries to integrate with programming languages and provide a uniform abstraction across languages.

An important question to ask is what functionality to expect from an event-based middleware. Its main focus is communication support for components in a distributed system. The middleware should efficiently and scalably assist the interaction between components. According to the idea of *distribution transparency* [CDK01], the complexity of distribution should be partially hidden from the programmer, thus simplifying the task of distributed application development. In addition to basic communication between components, the middleware should also support complex interaction patterns that are useful to programmers and relieve them from having to implement this functionality in the application. For example with respect to communication reliability, different levels of quality of service may be necessary to address the wide range of requirements in today's distributed applications.

Apart from communication, an event-based middleware, as with any form of middleware, must facilitate aspects of software engineering. This means that it should give the necessary support to the programmer, helping create a correct and well-structured distributed application. Issues, such as the usability of the middleware, become important factors that determine the degree to which an event-based middleware fulfils its task. Since a distributed application often must integrate with legacy components, interoperability of the middleware must be ensured. At all times, the complete life-cycle of a distributed application should be kept in mind. It must be easy to deploy the distributed application together with the middleware across the Internet. After deployment, the administration of the system has to be addressed, with the event-based middleware playing a substantial role in this. Finally, a distributed system evolves over its lifetime, and the middleware must be extensible to accommodate this — it is often too costly to redeploy an entire large-scale distributed system. For the purpose of this thesis, we will use the following definition of an event-based middleware.

Definition 3.1 (Event-based Middleware) *An* Event-based Middleware *is a middleware for large-scale distributed systems that provides scalable and efficient publish/subscribe communication between components. It also addresses traditional middleware requirements, such as usability, administrability, interoperability, and extensibility.*

In the next section, we return to the two application scenarios and use them to discuss some of the requirements for an event-based middleware. A better comprehension of the problems that a large-scale middleware aims to solve will lead to the development of design models that describe the data, the components, the routing, and the services of an event-based middleware.

3.2 Requirements

To specify the requirements for an event-based middleware, we will examine the two application scenarios introduced in Section 1.2 in more detail. The first application scenario, news story dissemination, is characterised by the large-scale widely-distributed deployment of the system. Scalability and efficiency are crucial in this context, but the heterogeneity of components also makes development challenging for the distributed applications programmer. In the second scenario, a ubiquitous computing environment, the complexity of the scenario comes from the vast amount of sensor data handled by the system. Components of the system can only cope with the data after preliminary filtering and processing by the middleware.

The requirements that will be derived from these two application scenarios can be grouped into two categories: First, there are requirements that emerge from the use of an information dissemination system, such as a publish/subscribe system. These requirements, such as scalability and expressiveness, are common to any large-scale application and are already addressed, to varying degrees, by current publish/subscribe systems. The second class of requirements are middleware requirements that are valid for any middleware, such as administrability and usability. Current middleware technology attempts to address them, and they are even more important in an event-based middleware because of the increased complexity of large-scale systems.

3.2.1 Scalability and Expressiveness

A standard requirement for any information dissemination system is scalability, which is the ability to support a large number of clients. The publish/subscribe communication model is intrinsically scalable because publishers and subscribers are only loosely-coupled, and the implementation of the event-based middleware must take advantage of this. However, this is challenging due to the trade-off between scalability and expressiveness [CRW99]. The expressiveness of a publish/subscribe implementation determines how fine-grained the selection mechanism is that information consumers use to characterise the kind of information that they want to receive. As a general rule, higher expressiveness means more state and processing, which reduces the overall scalability of the publish/subscribe system. Nevertheless, high expressiveness, as in a content-based publish/subscribe system, is desirable from a client's point of view. A client of the ubiquitous computing environment in the second application scenario prefers to express interest in data in as much detail as possible, delegating the data filtering burden to the producer-side.

To improve scalability, publish/subscribe systems have distributed implementations that spread the publish/subscribe service among multiple nodes. Efficient network-level communication primitives, such as IP multicast, can be used for distributing messages. However, relying on network-level support has the disadvantage that it restricts deployment only to supported networks. Moreover, the expressiveness of the system can suffer from the lack of features at the network-level, for example when providing content-based publish/subscribe over network-level group communication. A different approach is to leverage scalable, application-level communication abstractions, such as distributed hash tables, and advanced routing algorithms at the application-level to ensure expressiveness while maintaining scalability.

3.2.2 Reliability

Many distributed applications require strong reliability guarantees when using a middleware. For example, a customer in the first application scenario that pays a news agency for an uninterrupted stream of news reports expects the contract to be honoured even under some degree of failure. Similar to synchronous request/reply middleware that provides different invocation semantics and message-oriented middleware that supports persistent messages that can resist failure, an event-based middleware needs to provide different levels of reliability and be resilient to failure. Especially in a large-scale context, the failure of communication links and middleware components is not unusual. Resilience to failure must therefore be built into the event-based middleware from the beginning.

Different choices can be made to increase the robustness of an event-based middleware. Adding redundant middleware components improves availability but may be costly. As a general strategy, the amount of state maintained at middleware components should be minimised so that less state needs to be recovered after loss. This is non-trivial in a content-based publish/subscribe system, which may manage state about millions of subscriptions. For reliable service semantics, important state can be stored in persistent storage for simple recovery. Peer-to-peer techniques can assist in building resilient large-scale systems that still operate after a substantial number of their components have failed.

3.2.3 Administrability

An event-based middleware in itself is a complex distributed system with many components. Easy administrability of such a system is an important requirement, especially because any large-scale system may substantially evolve over its lifetime. When considering the first application scenario, it becomes obvious that deployment of a large-scale system is not straightforward. For instance, an event-based middleware may create a complex application-level overlay network, whose topology is not known in advance. The deployment not only determines the correct operation of the system, but also influences the efficiency of information processing. When new components are added to increase performance or availability, the event-based middleware has to adapt without significant amounts of human intervention.

The administration effort can be reduced by making components as autonomous as possible. Self-adapting systems can relieve the administrator from many decisions and thus facilitate the task of system administration. For this, peer-to-peer techniques are again a sensible choice because peer-to-peer systems are self-organising by definition. Adding a new component to the event-based middleware should only be a matter of "plugging it in". It is not realistic to assume a single system administrator with global knowledge of the entire system. Instead, the task of administration should be partitioned among multiple entities with partial responsibility.

3.2.4 Usability

A new middleware architecture will only be adopted if it is easy to use by the distributed applications programmer. The usability of a middleware is therefore an important factor and successful middleware platforms, such as CORBA and Java RMI, owe their success to features that facilitate their use. Current publish/subscribe systems are very primitive in this respect, whereas event-based middleware has to be designed with usability in mind. A deciding factor for the usability of an event-based middleware is how well it integrates with the programming language that is used to develop the distributed application. The data model of the event-based middleware should take advantage of programming language features.

Programming Language Integration

Synchronous request/reply middleware integrates naturally with an object-oriented programming language through proxy and remote objects. This programming model gives the illusion of distribution transparency to the programmer. Event-based middleware should follow similar abstractions that help integrate it with programming languages. The API for invoking its functionality should fit in with other services provided by the programming language. For example, a publish/subscribe service in Java can be viewed as an implementation of a distributed asynchronous collection [EGS00] that derives from the `Collection` datatype in the Java class libraries. Tighter integration with the programming language can be achieved by custom extensions to the language to support publish/subscribe functionality. However, this may result in tying the middleware down to a single programming language, which contradicts the requirement of interoperability stated below.

Handling of Event Data

Usability is also governed by how data handled by the middleware is represented in the programming language. There should be a natural mapping from middleware data to programming language objects. For example, in CORBA an IDL compiler creates language-specific stub code that automatically translates communication data into programming language objects. In an event-based middleware, data represented as events should be mapped to programming language objects so that it can be processed with language constructs. Event data should also be type-checked by standard language mechanisms, either statically at compile-time or dynamically at runtime. Type information associated with events can then assist in finding typing errors early on in the development process. Subscription data, which is used to specify a client's interest in event data, should also map to programming language entities and thus benefit from type-checking.

3.2.5 Extensibility

As mentioned before, an event-based middleware will evolve over the life-time of a distributed application. For example, in a large-scale system new communication protocols could be added and the event-based middleware would have to adapt to take advantage of them. When the communication requirements of middleware clients change, as may be the case after a new application is added to a ubiquitous computing environment, the event-based middleware should support the deployment of new services to address this. Features found in reflective middleware should therefore be part of any modern middleware, enabling it to inspect and modify its own components and their behaviour.

Extensibility in an event-based middleware can be achieved with a modular design. The middleware is partitioned into core components that are always needed and a set of domain-specific extensions that provide optional services. An advantage of a modular design is that components are only deployed when necessary, as opposed to a monolithic approach, in which the middleware cannot be decomposed. Customised deployment is important for restricted environments, such as embedded systems, where the memory footprint of the middleware has to be small.

Higher-Level Middleware Services

Middleware functionality is extensible with higher-level middleware services that are used by applications on demand. Examples are the CORBA object services mentioned in Section 2.1.1, which address a wide range of functionality from security to asynchronous messaging. Middleware services are only deployed when required by an application and new services are added to an existing middleware at runtime. Some services for an event-based middleware will be standard services as found in CORBA, whereas other services will be specifically targeted at a publish/subscribe environment with many-to-many communication semantics. An example of this would be a composite event detection service that enhances expressiveness of subscriptions by allowing the specification of complex event patterns.

3.2.6 Interoperability

With the heterogeneity of large-scale systems, interoperability between different forms of middleware becomes an important requirement. Connecting several types of event-based middleware is facilitated by the loose coupling of components in the publish/subscribe model. To further improve interoperability, an event-based middleware should be built on open standards that are platform- and language-independent. For example, XML as a messaging format simplifies the translation between different formats. The API exported by an event-based middleware should include bindings to several programming languages. This allows the distributed application programmer to choose the most convenient language for the implementation of a client.

Typically multiple deployments of an event-based middleware have a need to cooperate in order to exchange information. This can be achieved by forming an event federation [Hom02] between different event domains. In a federation, gateway components between domains translate the core publish/subscribe functionality that is supported among all event-based middleware deployments. A gateway component has to deal not only with structural but also with semantic heterogeneity, translating event data and metadata to achieve interoperability.

3.3 Design

We have outlined the requirements that we place on an event-based middleware. In this section we look at the design space for such a middleware and divide it into several design models that handle various aspects of the middleware. We use these design models to make different design decisions and point out the trade-offs with respect to our previous requirements. This discussion will also help establish common terminology to talk about event-based middleware, which will be used in the remainder of this dissertation. Previous work that structures an Internet-scale event notification system into design models was done by Rosenblum and Wolf [RW97]. However, our models substantially differ from previous ones as they explicitly acknowledge the distributed nature of an event-based middleware and put a stronger emphasis on middleware features.

The five design models are as follows: The data handled in the event-based middleware is described by an *event model*. The *component model* states the components that belong to the middleware. Communication issues of the middleware are part of a *routing model*, and a *reliability model* deals with resilience to failure. Finally, a *service model* describes how middleware extensions interact with the other models. We do not claim to survey the entire design space with these models, but they are a good starting point for an event-based middleware design.

3.3.1 Event Model

The main task of an event-based middleware is to disseminate data to all interested parties in a large-scale distributed system. The question of how data, and interest in that data, is expressed in the system is described by an *event model*. In general, data in an event-based middleware is represented in the form of *event publications* (or *events*). Event publications are manifested for routing and processing as *event publication messages* with a certain syntax and semantics. Publication messages are transfered over the network between distributed middleware components. The interest in events is formulated by *event subscriptions*. They can be regarded as queries over future events. The event-based middleware passes them around the system in the form of *event subscription messages*.

When designing an event model, its interaction with the data model of the programming language becomes important. The semantic gap between event and programming language data can be bridged by mapping publications and subscriptions onto programming language objects when they reach the clients of the event-based middleware. This mapping can be very natural if the event model respects the data model of the programming language. As a consequence, the notion of *datatypes* should be present in the event model. Strong type-checking can then verify that event data conforms to its type specification. In addition, most object-oriented programming languages support the concept of *type inheritance* to create more specialised subtypes from existing types. The same technique can be used to structure event data.

Event Publications. In our event model an event publication is an instance of an *event type* that has a *type schema* that describes the event type. This links the concept of a programming language type to an event. An event type contains an *event type name* and defines a set of *event attributes*, which are typed attribute/value pairs that hold the data in an event instance of this event type. Event publications can thus be regarded as record-like structures. An event type may also be associated with a *parent event type*, inheriting all its event attributes and entering into a subtyping relation with that type and all its ancestor types. The following definition makes these concepts more precise.

Definition 3.2 (Event Publication) *Every* event publication *e has an event type τ and belongs to the event space $\mathbb{E}$,*

$$(e : \tau) \in \mathbb{E}.$$

An event type *τ is a tuple, $\tau = (n_\tau, \tau_p, \{\tau_{a_1}, \tau_{a_2}, \dots, \tau_{a_k}\})$, where n_τ is an event type name, τ_p is a parent event type ($\tau \preceq \tau_p$), and $\{\tau_{a_1}, \dots, \tau_{a_k}\}$ is a set of event attribute types. An* event attribute type *τ_a is a pair, $\tau_a = (n_{\tau_a}, \tau_{\tau_a})$, where n_{τ_a} is an attribute name and τ_{τ_a} is its datatype. An event publication e is therefore a set of attribute/value pairs,*

$$e : \tau = \{(n_{a_1}, v_{a_1}), (n_{a_2}, v_{a_2}), \dots, (n_{a_k}, v_{a_k})\},$$

that conforms to the event type τ,

$$\forall (n_a, v_a) \in e : \tau.\ \exists (n_{\tau_a}, \tau_{\tau_a}) \in \tau.\ n_a = n_{\tau_a} \ \wedge\ \tau(v_a) \preceq \tau_{\tau_a},$$

where $\tau(v_a)$ is the datatype of v_a and $\preceq$ is the sub-typing relation.

Note that the introduction of event types into the event model to structure the event space $\mathbb{E}$ is incompatible with pure content-based publish/subscribe. Traditional content-based publish/-subscribe assumes a flat, unstructured event space, where events can posses arbitrary event attributes. Subscriptions contain content-based filtering expressions that select a subset of events from the event space. We argue that this unconstrained model is not realistic for distributed application development because, for software engineering reasons, it is desirable to structure the event space in order to cope with its size. The concept of types is an obvious way of doing this, which is why many pure content-based publish/subscribe systems include a special event attribute called `type` in all their event publications. By explicitly acknowledging event types in an event-based middleware, content-based routing algorithms can be made more efficient, as will be shown in Chapter 4.

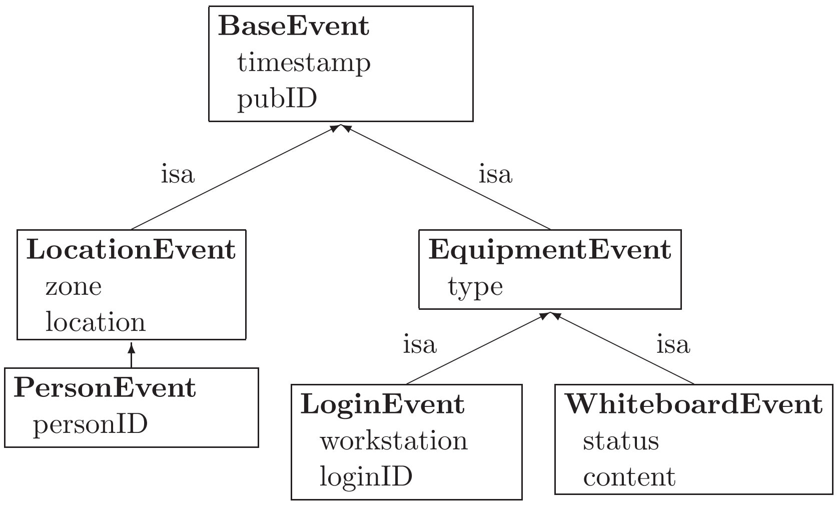


Figure 3.1: An example of an event type hierarchy

A further benefit of types in the event model is that type inheritance can be naturally integrated with the event-based middleware. New event types can be made more specialised after inheriting event attributes from ancestor types. This allows event types in an *event type hierarchy*, such as the example one shown in Figure 3.1 that could be used in the Active Office application scenario. Note that all event types are derived from a common `BaseEvent` ancestor that captures event attributes, such as an event timestamp `timestamp` and a publisher identifier `pubID`, that are found in all events in the system. However, sometimes a single ancestor event type may be too restrictive as it to some degree implies central administration of the event space.

When an event publication is expressed as a message, a language must be used that has a powerful enough type system to support the typing of events and event attributes. An example would be the ODMG Object Definition Language (ODL) that was used in the CEA to define schemas for event types [Pie00]. With the recent proliferation of XML as a standard format for semi-structured data, the more expressive XML Schema language [W3C01a] is a natural choice for publication messages in a heterogeneous distributed computing environment. It has a generic type system that supports the definition of new datatypes and provides type inheritance through restriction and extension.

Event Subscriptions. By the same reasoning as for publications, event subscriptions are typed as well. We propose a typed variant of content-based publish/subscribe called *type- and attribute-based publish/subscribe*. In this variant event subscribers express their interest in events in two stages. First, an *event type* (or a set of event types according to the event type inheritance relation) is chosen by the subscription. Then, a *content-based filtering expression* is provided that selects events of the specified event types that satisfy a set of predicates over their event attributes. Type-checking ensures that attribute predicates in event subscriptions comply with event type schemas. Type- and attribute-based publish/subscribe helps programmers decide which events from the event space are of interest to a client because the event type hierarchy is used to find relevant event types before specifying a content-based filter expression.

Definition 3.3 (Event Subscription) *An* event subscription s *has an event type* τ *and consists of a set of attribute predicates,*

$$s : \tau = \{p_1, p_2, \ldots, p_k\}.$$

An attribute predicate *p is a tuple, $p = (n_p, f_p, v_p)$, where n_p is an attribute name, f_p is a predicate filter function, and v_p is an attribute value. The subscription s conforms to an event type τ, $s : \tau$, if and only if*

$$\forall (n_p, f_p, v_p) \in s : \tau.\ \exists (n_{\tau_a}, \tau_{\tau_a}) \in \tau.\ n_p = n_{\tau_a}\ \wedge\ \tau(v_p) \preceq \tau_{\tau_a}$$

holds, where $\tau(v_p)$ is the datatype of v_p and $\preceq$ is the sub-typing relation.

An event matches a subscription if it is of the correct event type and satisfies all attribute predicates specified in the subscription. Only the conjunction of attribute predicates is allowed in subscriptions because disjunctive predicates can be emulated through multiple subscriptions. Note that supertype subscriptions are supported because the subtyping relation determines if the event types of the event publication and subscription are compatible. As in other publish/-subscribe systems, we also include a coverage relation between subscriptions. A subscription is covered by another subscription if the former is more specific than the latter. In this case the second subscription is also matched by any event that is matched by the first subscription. A content-based routing algorithm can then benefit from this relation between subscriptions to reduce the state stored in the event-based middleware.

Definition 3.4 (Subscription Coverage) *An event e is* covered *by (or* matches*) a subscription s,*

$$e \sqsubseteq s,$$

if and only if

$$\forall p \in s : \tau_s.\ \exists a \in e : \tau_e.\ \tau_s \preceq \tau_e \wedge a \sqsubseteq p$$

holds. An event attribute $a = (n_a, v_a)$ is covered *by (or* matches*) an attribute predicate $p = (n_p, f_p, v_p)$,*

$$a \sqsubseteq p,$$

if and only if

$$n_p = n_a \wedge f_p(v_p, v_a)$$

holds. A subscription s_1 is covered *by another subscription s_2,*

$$s_1 \sqsubseteq s_2,$$

if and only if

$$\forall e \in \mathbb{E}.\ e \sqsubseteq s_1 \Rightarrow e \sqsubseteq s_2.$$

Different alternatives for stating predicate filter functions are possible, which then influence the expressiveness of subscriptions in the event-based middleware. Since subscriptions are queries over future events, publish/subscribe systems often adopt database query languages, such as SQL92 [SQL92] used by the JMS specification. This leads to very powerful filters and therefore makes the evaluation of coverage relations computationally expensive, because advanced matching algorithms construct efficient indexing structures by discovering commonality between filters. An opposite approach is to have very restricted predicate filter functions and only permit equality predicates for template-based filtering.

Since we argued for XML as a format for writing event publication messages, a uniform approach should be taken for event subscriptions in subscription messages. A single publication message defined in XML Schema can be queried with XPath [W3C99b]. The XPath language includes a wide range of built-in filter functions and follows the XML Schema type model. A fully-featured query language for XML is XQuery [W3C03d], but detecting coverage between two arbitrary XQuery expressions is non-trivial.

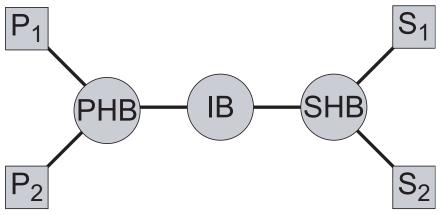


Figure 3.2: Illustration of the component model

3.3.2 Component Model

We argued that a middleware for large-scale systems must have a distributed implementation in order to be scalable and fault-tolerant. This means that the middleware consists of a number of components that may differ in their purpose and run on separate nodes in the network. A *component model* describes these components and the relation between them. As a general principle, the load between components should be equally balanced so that no component can become a performance bottleneck or a single point of failure for the entire system. Adding multiple redundant instances of the same component can improve the robustness of the system.

In our component model, we introduce two kinds of components for an event-based middleware, *event brokers* and *event clients*. Event brokers implement the entire functionality of an event-based middleware and provide a service to event clients. To use the event-based middleware, event clients have to connect to at least one event broker. Event clients come in two flavours, *event publishers* that publish and *event subscribers* that subscribe to events. Since event clients closely interact with the application, they are not language-independent, whereas event brokers are. An event-based middleware with event brokers and event clients is shown in Figure 3.2. Next we describe the different types of event brokers using terminology that follows that used by Gryphon in Section 2.2.2.

Event Brokers. Event brokers are the main components of an event-based middleware. A single event broker constitutes a complete implementation of the middleware, but usually multiple event brokers are deployed together. Event brokers cooperate with each other by forming an overlay routing network and execute a content-based routing algorithm for events, such as the algorithm used by Hermes in Chapter 4. It is assumed that event brokers in the overlay routing network all follow the same protocol to provide a service to event clients connected to the middleware. As a consequence, less trust needs be assigned to event clients because a misbehaving client can be removed from the middleware without affecting the service provided to other clients. In contrast, a malicious event broker can substantially degrade the service of the middleware and cause damage to all clients.

An event broker that has one or more event publishers connected is called a *publisher-hosting broker* (PHB). An event broker becomes a *subscriber-hosting broker* (SHB) if it is maintaining a connection to event subscribers. Event clients connected to event brokers consider them to be their *local event brokers*, as they function as local entry points to the event-based middleware. An event broker can be both subscriber-hosting and publisher-hosting, or neither, in which case it is called an *intermediate broker* (IB). An event broker in the system only knows about a subset of all existing event brokers, its *neighbouring event brokers*, and can only send messages to these event brokers, as dictated by the routing algorithm. The relation of neighbouring event brokers forms the *overlay broker network* of the event-based middleware. These concepts are illustrated in Figure 3.2 and formalised below.

Definition 3.5 (Event Broker) *An* event broker $B \in \mathbb{B}$ *from the set of all event brokers* $\mathbb{B}$ *maintains a tuple,*

$$B = (C_P, C_S, N_B, \mathcal{S}),$$

where C_P *is a set of event publishers and* C_S *is a set of event subscribers that this event broker is hosting,* N_B *is a set of neighbouring event brokers, and* $\mathcal{S}$ *is routing state kept at the event broker. An event broker is* publisher-hosting *if and only if* $C_P \neq \emptyset$*, and* subscriber-hosting *if and only if* $C_S \neq \emptyset$*. If* $C_P = C_S = \emptyset$ *holds, then the event broker is an* intermediate broker*.*

The graph forming the overlay broker network*,* $G_O = (\mathbb{B}, N_G)$*, consists of all event brokers* $\mathbb{B}$ *and the relation* N_G *of neighbouring event brokers sets,*

$$N_G = \{(B, N) \mid \forall B \in \mathbb{B}\ \forall N \in N_B.\ B = (C_P, C_S, N_B, \mathcal{S})\}.$$

Event Publishers. An event publisher is a client component that produces event publications and passes them to the event-based middleware for dissemination. It maintains a connection to at least one local event broker and does not possess any middleware functionality by itself. This has the advantage that event publishers are light-weight components with modest resource requirements, making it possible to deploy them in embedded or mobile devices. Technically, an event publisher is outside the event-based middleware model because it is under the control of the distributed system programmer. Therefore, its correct behaviour cannot be guaranteed and the event-based middleware must take account of this.

Event publishers use a client interface exported by event brokers to request middleware functionality, such as managing event types and publishing events. Due to the heterogeneity of the environment, this interface should be language-independent. Since this interface only handles the communication of the event client with its local event broker, it may be synchronous or asynchronous. An asynchronous interface has the advantage that event publishers do not need to wait for a response when publishing events under best-effort semantics. Since inter-broker communication is already XML-based, web services, as described in Section 2.1.3, are a natural choice for communication between event publishers and event brokers. Internally the event publisher then exports a language-dependent interface to the application.

Definition 3.6 (Event Publishers) *An* event publisher $P \in \mathbb{C}$ *from the set of all event clients* $\mathbb{C}$ *maintains a pair,*

$$P = (L_P, \mathcal{S}),$$

where L_P *is a set of local event brokers that it has connections with and* $\mathcal{S}$ *is its state.*

Event Subscribers. The second type of client component are the event subscribers, which subscribe to events and consume event publications, passing them on to the application. Analogously to event publishers, event subscribers are light-weight components and maintain connections to local event brokers to request middleware functionality. Their main interface operations deal with the management of event types, the specification of event subscriptions, and the notification of matching event publications.

Unlike event publishers, event subscribers receive asynchronous notifications from their local event brokers whenever an event is published that matches one of their subscriptions. For this, they export an asynchronous callback interface to a local event broker. This interface must be asynchronous to achieve a timely notification of events. Handling these callbacks makes the implementation of event subscribers more complex than that of event publishers.

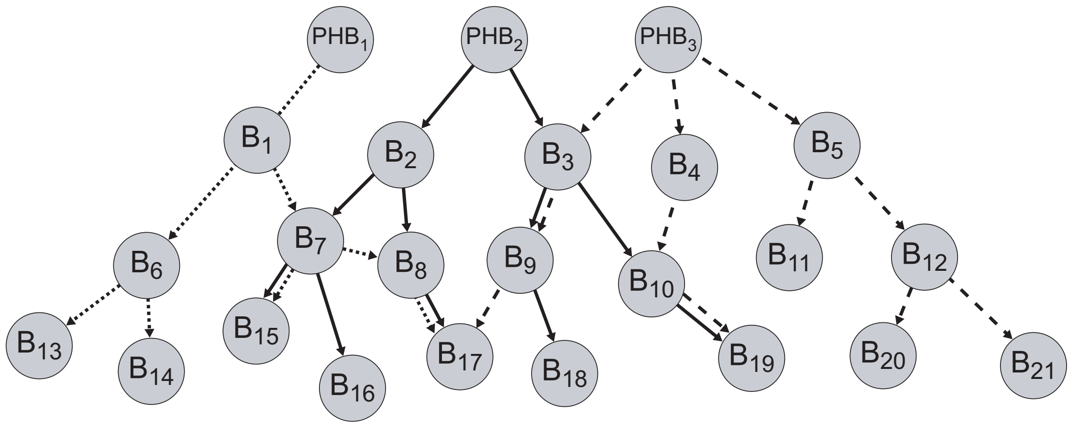


Figure 3.3: Event dissemination trees in a publish/subscribe system

Definition 3.7 (Event Subscriber) *An* event subscriber $S \in \mathbb{C}$ *from the set of all event clients* $\mathbb{C}$ *maintains a pair,*

$$S = (L_S, \mathcal{S}),$$

where L_S *is a set of local event brokers that it has connections with and* $\mathcal{S}$ *is its state.*

3.3.3 Routing Model

The distributed nature of an event-based middleware means that events have to be routed in a network. Informally, events that are produced by event publishers enter the system at publisher-hosting brokers and are disseminated to all subscriber-hosting brokers that have event subscribers with matching event subscriptions. In content-based routing, events are routed depending on their content, and routing decisions are made with respect to previously submitted subscriptions. A trace-based semantics of the desired behaviour of a publish/subscribe system can be found in [Müh02]. To achieve this behaviour, a *routing model* for an event-based middleware describes the algorithms that govern the propagation of event publications and subscriptions in the network and the routing state that is maintained at the middleware components. Since it is usually assumed that event publications are more common than subscriptions, routing state should be set up in response to subscriptions, so as to facilitate the routing of publications.

Most publish/subscribe systems construct one or more *event dissemination trees* that are used for content-based routing. As shown by the three event dissemination trees in Figure 3.3, the root is a source of events, such as a publisher-hosting broker, and the nodes correspond to subscriber-hosting brokers with event subscribers and potentially matching event subscriptions. Event subscriptions propagate up the tree and establish filtering state at the nodes of the tree. This filtering state then controls the downwards flow of events and ensures event types are only delivered to interested event subscribers. The strategy of *source-side filtering* states that filtering state is created as close to event producers as possible so that unnecessary events can be discarded without wasting resources such as network bandwidth or processing power. There is either a single event dissemination tree that is statically shared among all event producers, or separate trees that are created dynamically as new event producers join the publish/subscribe system. The disadvantage of a single event dissemination tree is that it is harder to support multiple sources of events. Therefore, we use a *forest of event dissemination trees* that is rooted at publisher-hosting brokers and supports the efficient dissemination of events of a given type published by each publisher.

The construction of an event dissemination tree can be assisted by *event advertisements* that help establish routing state in the system. An event advertisement is an indication sent by an event producer that it will publish events of a particular event type. Advertisements also support a coverage relation with event subscriptions and other advertisements that is evaluated by the routing algorithm.

Definition 3.8 (Event Advertisement) *An* event advertisement *a has an event type τ,*

$$a : \tau.$$

A subscription $s : \tau_s$ is covered *by (or* matches*) an advertisement $a : \tau_a$,*

$$s \sqsubseteq a,$$

if and only if

$$\tau_a \preceq \tau_s$$

holds, where $\preceq$ is the sub-typing relation. Coverage between advertisements is analogous.

Advertisements sent by publisher-hosting brokers and subscriptions sent by subscriber-hosting brokers have to join in the network in order to set up a path from event producers to event consumers. All those paths form a forest of event dissemination trees. A simple approach to join advertisements and subscriptions is to make either globally known to all nodes in the network. However, as argued below, global state in routing algorithms reduces scalability. Instead, the Hermes routing algorithms, which will be presented in Chapter 4, use the concept of a rendezvous node, as a well-known point in the network at which advertisements and subscriptions meet, resulting in a forest of event dissemination trees.

After construction of an event dissemination tree, routing of events at each node has two stages: First, an event publication is matched against event subscriptions that were submitted by locally-hosted event subscribers (*local event matching*). Then, the event publication is matched against remote event subscriptions so that a routing decision can be made that establishes the next hops for the event (*remote event matching*). A large body of work [ASS+99, FJL+01] addresses the issue of efficient local event matching using database indexing techniques.

For scalable routing, it is important that the routing decision for an event does not require global knowledge of, say, all event subscribers or all event brokers in the system. Instead, only state about a subset of all brokers, such as the set of neighbouring event brokers, should suffice. Messages should also not be globally broadcast to all nodes in order to create common state. Such broadcasts defeat scalability as they introduce an unknown delay until a message has been successfully processed by all nodes. Moreover, broadcasts limit the robustness of the event-based middleware since a single node failure may cause the entire application-level broadcast to fail.

The shape of an event dissemination tree determines important properties of the publish/-subscribe system, such as efficiency of event dissemination and resilience to failure. A tree can be constructed in two ways, at the application-level or at the network-level. In application-level routing, the routing of event publications is done entirely by application-level components running at nodes, such as event brokers, whereas network-level routing uses efficient group communication primitives and router support in the network. Next we will discuss the trade-offs associated with both approaches.

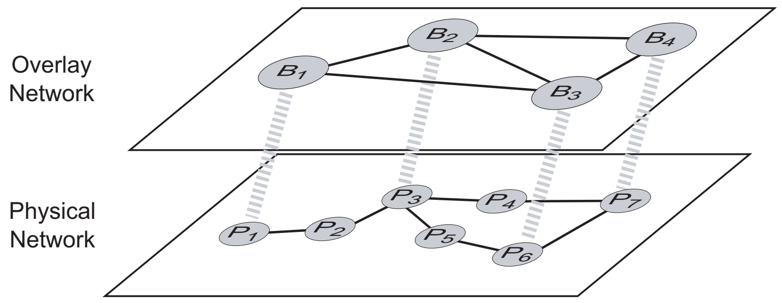


Figure 3.4: Mapping an overlay network onto a physical network

Application-Level Routing

Application-level routing only requires full unicast connectivity between nodes in the network, such as provided by IP communication on the Internet. Here the event dissemination tree is built as a part of an *overlay broker network* on top of a physical network topology. Its nodes consist of event brokers that route event publications and its edges are defined by the relation of neighbouring event brokers. Routing an event to all interested subscriber-hosting brokers involves the traversal of multiple application-level hops but does not need any special network features.

An advantage of application-level routing is the flexibility of the routing algorithms that can be implemented. Sophisticated routing decisions can be made at every hop as the entire processing is implemented in the event-based middleware. In particular, robust routing with resilience to failure is easier to achieve at the application-level because the overlay network benefits from redundant routes and middleware components. Another benefit is that the deployment of the middleware does not rely on the existence of any special network features, such as IP multicast, which may not be available everywhere.

The price to be paid for this flexibility is a reduction in routing efficiency. As shown in Figure 3.4, a single hop in the overlay network may result in multiple hops in the underlying physical network topology. In this example, event broker B_1 has a neighbouring event broker B_3 in the overlay network that is geographically far away in the physical network — the shortest path in the physical network is 4 hops. Any communication between these two brokers will be expensive in terms of network utilisation. Costs should be associated with links in the overlay topology, reflecting the actual costs of routing messages in the physical network. The quality of the mapping of the overlay network down to a physical topology will determine the efficiency of routing in the event-based middleware. In Chapter 5 we will introduce cost metrics to evaluate the efficiency of an overlay broker network.

Network-Level Routing

To avoid the penalty of application-level routing, the network can assist in establishing event dissemination trees without the indirection of an overlay broker network. In network-level routing, the event-based middleware takes advantage of advanced network features that support the routing of events in a publish/subscribe scenario. For instance, many current networking technologies provide multicast communication to send a single message efficiently to members of a group. A multicast routing algorithm constructs a dissemination tree with network routers as nodes, taking the physical topology of the network into account. Networking technologies deployed in specialised domains, such as mobile ad-hoc networks or sensor-rich environments, often come with efficient group communication built into their core routing protocols.

In active networks [TSWM97], routers are programmable entities that can execute portions of a content-based routing algorithm. This opens up the possibility of delegating parts of the event-based middleware into the network itself and thus improving the efficiency of event dissemination [CBP+02]. Schemes, such as *Generic Router Assist* (GRA) [CST01], allow filter expressions that derive from event subscriptions to be installed at routers in the network, which can then perform content-based routing of events with a comparable efficiency to normal, address-based routing. This new style of networking could be viewed as *content-based networking* [CW01] that departs from the traditional concept of address-based routing of datagrams.

However, an event-based middleware that heavily relies on sophisticated network-level features is difficult to deploy in general purpose networks. Traditional network-level group communication primitives, such as IP multicast, are often hard to integrate with multiple information sources and content-based routing, in which the notion of static membership of a limited number of groups does not exist. As a result, efficiently mapping event subscribers to multicast groups remains an unsolved problem [BCM+99, RLW+03]. Our event-based middleware, Hermes, therefore routes events at the application-level, and will be described in Section 4.3.

3.3.4 Reliability Model

The *reliability model* handles fault-tolerance mechanisms to cope with failure in the event-based middleware. Reliability has two aspects: When clients use a best-effort service for event dissemination, they expect the middleware to exhibit robustness against failure. A robust system attempts to reduce degradation in service caused by failure to a minimum, otherwise best-effort semantics become unusable in the face of frequent data loss. The second aspect of reliability is when it is explicitly demanded by clients through a quality of service (QoS) specification. For example, an event client may publish events under *guaranteed delivery* semantics, so that it becomes the responsibility of the middleware to correctly deliver those events to event subscribers, even under failure.

When designing mechanisms that handle failure, it is necessary to have a failure model of the supported types of failure in the event-based middleware. We distinguish between two common types of failure, *network failure* of the physical network of links and routers, and *middleware failure* of components of the middleware. Next we discuss options for addressing these types of failure in an event-based middleware. We also outline mechanisms that provide stronger reliability guarantees than best-effort semantics to event clients using persistent events. In Section 4.3.7 we will present our implementation of the reliability model in Hermes.

Network Failure. Network failure can be modelled as link failure that results in the short- or long-term inability of event brokers to contact neighbouring event brokers. It can be caused by a failure at the network-level, such as a router fault. With application-level routing, the robustness of the middleware can be increased by handling network failure in the overlay network [ABKM01]. The overlay network has redundant paths between event brokers so that a different path can be chosen when a given physical link — and thus any logical links that depend on that link — is down. This means that the overlay network topology has to change because of failure in the physical network. If the overlay network is managed using peer-to-peer techniques, the adaptation of routing can occur transparently to the event-based middleware, avoiding loss of events under best-effort semantics. To cope with the failure of local event brokers, event clients can maintain redundant connections with multiple event brokers in the event-based middleware.

Middleware Failure. Failure of an event-based middleware component means that an event broker or client has stopped working and thus cannot participate in the operation of the middleware any more. Usually component failure in a distributed system is discovered with a *heartbeat protocol* that exchanges messages among components at regular intervals when there is no other communication. When an event broker fails, state about event dissemination trees may be lost, causing gaps in trees. The event broker may also lose state about hosted event clients. Note that failure of an event client may also be voluntarily, if it decides to temporarily disconnect from the middleware. For example, in a mobile environment event clients detach from and reattach to event brokers while roaming.

A crucial part of the reliability model is the repair of event dissemination trees after failure of event brokers. Rather than recovering an old tree, it is often easier to build a new event dissemination tree by resending event advertisements and subscriptions. Since the overlay broker network will have adapted to the failure, the resent messages will take different routing paths and hence create state in new event brokers, circumventing the failed components and establishing again a complete tree with filtering state. However, this still leaves the problem of garbage-collecting old state from event brokers after the repair of an event dissemination tree.

State in the middleware can be *hard* or *soft state*. The difference is that hard state never expires, as opposed to soft state, which has to be periodically refreshed to persist. This so-called lease-based approach has the advantage that old state at event brokers does not have to be garbage-collected, because old state will expire automatically when it is no longer refreshed by other event brokers.

Persistent Events. The methods outlined above improve the robustness of best-effort semantics, but may still lead to event loss. For event clients requiring stronger guarantees, event publications have to be preserved on persistent storage to survive failure of the event-based middleware. *Persistent events* will be eventually delivered to all interested subscribers once the operation of the middleware can resume. For reliable event delivery, additional algorithms are necessary that can resend events after recovery of the system or the reconnection of an event subscriber in a mobile environment.

Events can be stored in a persistent log at every event broker during routing. Routing can then resume from the last hop before the failure, but the performance of the middleware suffers from the frequent accesses to persistent storage. If failures are isolated occurrences, a better solution is to keep an authoritative copy of event publications at the original publisher-hosting broker that can then resend events lost during propagation [BSB$^+$02]. Intermediate brokers cache events for a certain period to quickly satisfy requests for retransmission. Once the event has been delivered and acknowledged by all event subscribers, it is removed from the authoritative log at the publisher-hosting broker. Mobile event subscribers can use the log at the publisher-hosting broker to request the replay of events missed during disconnection [BZA03].

3.3.5 Service Model

Extensibility of middleware was identified as an important requirement. Therefore we introduce a *service model* that describes the interaction of middleware services with the remaining design models. Only core communication functionality is part of the event-based middleware by default and thus implemented by event brokers. Extensions not targeted at all applications are included as services and used on demand. This section gives an overview of three useful services for an event-based middleware. We infer the required support in the architecture of the event-based middleware to realise these services and outline their interfaces with the rest of the middleware.

A wide range of higher-level services are conceivable for an event-based middleware. The list of CORBA object services in Section 2.1.1 gives an idea of useful services in a distributed computing environment. Next we highlight the features of an event-based middleware that are necessary to support three different services. The first service, *congestion control*, prevents congestion in the event-based middleware. The second service is *composite event detection*, which is a publish/-subscribe-specific service that extends the expressiveness of middleware. Finally, a *security* service adds access control to the event-based middleware.

Congestion Control

Congestion at event brokers or in the overlay network can lead to the collapse of event dissemination. A congestion control service solves the congestion problem by restricting the publication rate of events when resources are scarce and congestion could occur in the event-based middleware. When a system is significantly overprovisioned or events can be discarded without negative impact, congestion control in the middleware is unnecessary. Since not all distributed applications require congestion control, it is implemented as an optional middleware extension.

A congestion control service focuses on the event brokers in the middleware. The event brokers monitor for symptoms of congestion and adapt their processing appropriately. A modular design of the event broker allows the insertion of a congestion control module into the data path taken by event publications. Moreover, new types of messages may need to be added to the middleware for notification of congestion bottlenecks in the system. These messages are intercepted by the congestion control module. The marginal involvement of event clients in overlay network congestion control means that a plug-in interface at event brokers is sufficient for this service extension. In Chapter 6 our congestion control service will be described in detail.

Composite Event Detection

Composite event detection enhances the expressiveness of the event-based middleware with complex subscriptions that allow the specification of interest in patterns of events. These composite events are detected by dedicated composite event detectors, thus simplifying the implementation of event clients with complex interests in events. Many applications do not need complex subscriptions, which is why composite event detection is a good example of an optional middleware service that is only meaningful in the publish/subscribe model.

This service is implemented by composite event detectors, which are additional middleware components distributed throughout the system. Composite event detectors may be co-located with event brokers or deployed as stand-alone components. From the perspective of the service model, they behave like event clients and use the event-based middleware to communicate with each other. Since composite event detection leverages publish/subscribe functionality to provide a more expressive publish/subscribe service, this service can be understood as an additional layer on top of the event-based middleware with relative independence of the internals of event brokers. A full description of a composite event service will given in Chapter 7.

Security

The final service example makes an event-based middleware secure. Adding security to a middleware platform is challenging because security is a cross-cutting issue that has to be integrated from the start, rather than being added retrospectively. A security service restricts access to the perimeter of the event-based middleware, making access control decisions at local event brokers when clients request middleware functionality. In addition, confidentiality has to be guaranteed through secure inter-broker connections and separate levels of trust for event brokers.

A security module at the local event brokers handles the authentication and authorisation of event clients. It must intercept API calls by event clients and make access control decisions. Moreover, a cryptographic engine is inserted into the data path at event brokers in order to encrypt and decrypt event publications. It also has to perform cryptographic key management and ensure the security of connections in the overlay broker network. When event publications can contain encrypted data, the coverage relation is affected, as well. From this, it becomes obvious that a security service depends on a flexible and pluggable architecture so that modules can be replaced with secure versions. The full extent of adding security to an event-based middleware will be shown in Chapter 8.

3.4 Summary

In this chapter the notion of an event-based middleware was refined through a number of design models. We started with an overview of the requirements for an event-based middleware. The trade-off between scalability and expressiveness was discussed, and fault-tolerance was identified as an essential requirement for large-scale systems. In addition, we stressed the importance of administrability and usability of a middleware, such as good programming language integration, as this ensures that a new middleware platform will be adopted by programmers. Finally, the heterogeneity of large-scale systems requires extensible and interoperable designs.

The specification of requirements helped structure the design space for an event-based middleware into five design models. An event model for the data handled in the middleware was developed. This model features event typing as an important concept in an event-based middleware and a formalisation of event publications and subscriptions. The components of the event-based middleware are part of a component model, with event brokers providing a service to event clients. Algorithms for the dissemination of events belong to the routing model. We made a distinction between application-level and network-level routing for event dissemination, and chose the more flexible content-based routing of events at the application-level. To facilitate the construction of event dissemination trees, event advertisements were included in the model. A reliability model addressed the requirement of fault-tolerance with mechanisms to cope with certain types of failure in the middleware. Finally, we introduced a service model for middleware services and discussed techniques for adding a service to the middleware with three example services.

The design models from this chapter serve as a first step towards a concrete architecture for an event-based middleware that satisfies our requirements. In the next chapter, a scalable, event-based middleware architecture, called HERMES, will be presented that constitutes a named selection from this design space. More detail on extensions to the core HERMES middleware in the form of middleware services according to the service model will also be given in the remaining chapters of this dissertation.

4 Hermes

This chapter presents Hermes [PB02], a scalable, event-based middleware architecture that facilitates the building of large-scale distributed systems. Hermes has a distributed implementation that adheres to the design models developed in the previous chapter. It is based on an implementation of a peer-to-peer routing layer to create a self-managed overlay network of event brokers for routing events. Its content-based routing algorithm is highly scalable because it does not require global state to be established at all event brokers. Hermes is also resilient against failure through the automatic adaptation of the overlay broker network and the routing state at event brokers. An emphasis is put on the middleware aspects of Hermes so that its typed events support a tight integration with an application programming language. Two versions of Hermes exist that share most of the codebase: an implementation in a large-scale, distributed systems simulator, and a full implementation with communication between distributed event brokers.

We begin with an overview of Hermes in the next section, followed by a description of its architecture in Section 4.2. In Section 4.3 we explain the novel routing algorithms used by Hermes for content-based routing on top of a distributed hash table. Our prototype implementation of Hermes is discussed in Section 4.4, justifying our decisions with respect to the event-based middleware design models in more detail.

4.1 Overview

A primary feature of the Hermes event-based middleware is scalability, as it is targeted at the development of large-scale distributed systems. Hermes includes two content-based routing algorithms to disseminate events from event publishers to subscribers. The *type-based routing algorithm* only supports subscriptions depending on the event type of event publications. It is comparable to a topic-based publish/subscribe service but differs by observing inheritance relationships between event types. The second algorithm is *type- and attribute-based routing*, which extends type-based routing with content-based filtering on event attributes in publications. In both algorithms, event-type specific advertisements, introduced in Section 3.3.3, are

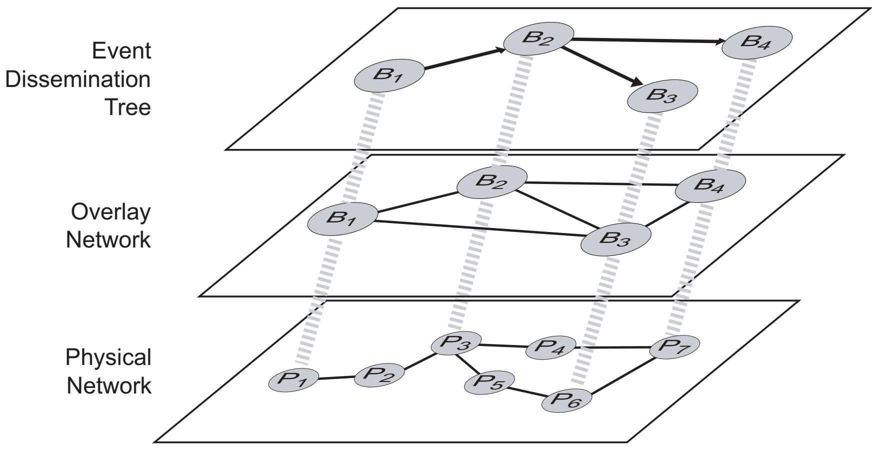


Figure 4.1: Layered networks in HERMES

sent by publisher-hosting brokers to set up routing state. Advertisements are not broadcast to all event brokers, but instead event brokers can act as special rendezvous nodes that guarantee that event subscriptions and advertisements join in the network in order to form valid event dissemination trees. The trade-offs associated with the two routing algorithms will be investigated in Section 4.3.

Both routing algorithms use a distributed hash table to set up state for event dissemination trees. The distributed hash table functionality is implemented by a peer-to-peer routing substrate, called PAN, formed by the event brokers in HERMES. PAN is an extended implementation of the Pastry routing substrate from Section 2.3.1. The advantage of such peer-to-peer overlay network are threefold: first, the overlay network can react to failure by changing its topology and thus adding fault-tolerance to HERMES. Second, the peer-to-peer routing substrate that manages the overlay network is responsible for handling membership of event brokers in a HERMES deployment. Third, the discovery of rendezvous nodes, which must be well-known in the network, is simplified by the standard properties of the distributed hash table.

The three layers of networks in HERMES are illustrated in Figure 4.1. The bottom layer is the physical network with routers and links that HERMES is deployed in. The middle layer constitutes the peer-to-peer overlay network that offers a distributed hash table abstraction. The top layer consists of multiple event dissemination trees that are constructed by HERMES to realise the event-based middleware service. When a message is routed using the peer-to-peer overlay network, a callback to the upper layer is performed at every hop, that allows the event broker to process the message by altering it or its own state.

In addition to scalable event dissemination, HERMES supports event typing, the creation of event type hierarchies through inheritance, and generic, supertype event subscriptions. This enhances its integration with current object-oriented programming languages such as Java or C++.

4.2 Architecture

Mirroring the layered network structure, the architecture of HERMES has six layers, shown in Figure 4.2. Each layer builds on top of the functionality provided by the layer underneath and exports a clearly defined interface to the layer above. Apart from that, the layers are independent of each other. A layered architecture for a communications system has the advantage that each

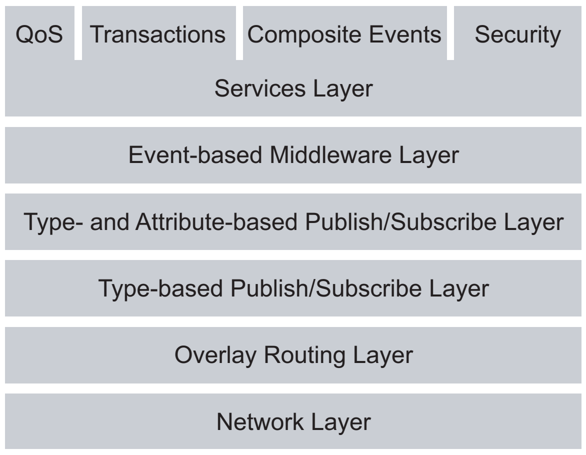

QoS
Transactions
Composite Events
Security
Services Layer
Event-based Middleware Layer
Type- and Attribute-based Publish/Subscribe Layer
Type-based Publish/Subscribe Layer
Overlay Routing Layer
Network Layer

Figure 4.2: Overview of the Hermes architecture

layer can have its implementation easily replaced by a different implementation if necessary. For example, if a more efficient implementation of a distributed hash table becomes available, Hermes can benefit from this without major modification. Since Hermes is implemented by the event brokers, its layered structure is also reflected in the implementation of an event broker. Next, we will describe the role of each layer, starting with the lowest one.

Network Layer. The lowest layer is the network layer that represents the unicast communication service of the underlying physical network. We assume that Hermes is deployed in a network with full unicast connectivity between nodes, such as the Internet. No other network-level services, such as group communication primitives, are necessary.

Overlay Routing Layer. The overlay routing layer implements an application-level routing algorithm that provides the abstraction of a distributed hash table. A peer-to-peer implementation of this layer is chosen for reasons of scalability and robustness. This layer takes application-level nodes, which are Hermes event brokers, and creates routing state in order to hash keys to nodes. It also handles the addition, removal, and failure of nodes in the overlay network. The topology of the overlay routing layer is optimised with respect to a proximity metric of the underlying physical network.

Type-based Publish/Subscribe Layer. The type-based publish/subscribe layer exports a primitive type-based publish/subscribe service on top of the distributed hash table established by the previous layer. Type-based routing supports subscriptions according to an event type, and observes the inheritance relationships between event types. Event dissemination trees are then created with the help of rendezvous nodes in the system. Trees are also repaired by retransmitting messages after state at event brokers has been lost.

Type- and Attribute-based Publish/Subscribe Layer. This layer extends the type-based service with content-based filtering on event attributes. The same rendezvous node mechanism is used for the construction of event dissemination trees. However, the trees are annotated with filtering expressions derived from the type- and attribute-based subscriptions. These filtering expressions are placed at strategic locations in the network, usually as close to event producers as possible in order to discard unnecessary events as soon as possible.

Event-based Middleware Layer. At this layer event-based middleware functionality is added to the content-based publish/subscribe system of the previous layers. Typing information

Returns	API Call	Parameters
`void`	*connectBroker*	`(EventBroker broker, Credentials c, EventBroker bootstrapBroker)`
`void`	*disconnectBroker*	`(EventBroker broker, Credentials c, EventBroker disconnectBroker)`
`void`	*disconnectBrokerFromAll*	`(EventBroker broker, Credentials c)`
`Set`	*getNeighbouringBrokers*	`(EventBroker broker, Credentials c)`

Table 4.1: The Hermes event broker API

is maintained at the rendezvous nodes so that event publications and subscriptions can be automatically type-checked by Hermes. The event-based middleware layer also extends the API used by event clients to invoke Hermes.

Services Layer. The services layer is a set of pluggable extensions to the event-based middleware layer. It allows Hermes to provide a wide range of higher-level middleware services. For example, different guarantees of publication and subscription semantics can be supported by a QoS module at the services layer. Another service may deal with composite event detection or transaction support. Services may violate the strict layering of the architecture and obtain direct access to lower layers if this is necessary for their functionality.

Having described the general architecture of Hermes, we will concentrate on the components of Hermes. Hermes closely follows the component model developed in Section 3.3.2, dividing the event-based middleware into event brokers and event clients.

4.2.1 Event Brokers

A Hermes event broker implements all layers except the network layer. The overlay routing layer is a Pan node, which represents a single node in the overlay network that is capable of receiving values for keys in the distributed hash table. On top of this, the two publish/subscribe routing layers export a client interface that event publishers and subscribers use. The event-based middleware layer of a Hermes event broker supports an external API that is used for administration of Hermes.

An excerpt of the external Hermes event broker API is shown in Table 4.1. Note that exceptions raised by API calls in response to error conditions are ignored for ease of presentation. The event broker API is as simple as possible in order to comply with the requirement for administrability from Section 3.2.3. All API methods take the event broker identity and authentication credentials as parameters that are used for an access control decision, as will be described in Chapter 8. The main task of this API is the establishment of the overlay broker network and thus the maintenance of the set of neighbouring event brokers. For this, a new event broker that wishes to join an existing Hermes deployment invokes the `connectBroker` method, passing the identity of a `bootstrapBroker` as a parameter.

The bootstrapping event broker is only used as an initial entry point to the overlay network. The join protocol of the distributed hash table, as outlined in Section 2.3.1, finds an optimal set of neighbouring event brokers for the new broker with respect to the proximity metric. This means that the quality of the overlay network does not depend on the choice of bootstrapping event broker as that broker may not end up in the set of neighbouring event brokers. After the new event broker has successfully connected to the bootstrapping broker, its set of neighbouring event brokers will be populated with entries by the peer-to-peer routing substrate PAN. The number of neighbouring event brokers for every event broker is a tunable parameter of PAN that follows a trade-off between resilience and statefulness of the network.

A call to `disconnectBroker` disconnects an event broker. This is achieved by flagging the event broker to be disconnected as failed, which causes the overlay routing substrate to remove the broker and restore any invariants of the distributed hash table, adjusting the set of neighbouring event brokers. Disconnecting particular event brokers in HERMES is mainly used for system maintenance and policy enforcement so that, for example, an event broker in one administrative domain may not have connections to brokers in a second domain. To leave a HERMES network of event brokers, a call to `disconnectBrokerFromAll` removes an event broker entirely from the overlay network and restores any invariants by following the leave protocol of the distributed hash table. Finally, the method `getNeighbouringBrokers` returns the set of neighbouring event brokers that an event broker is currently connected to.

When an event broker is a member of a HERMES network, it can become part of one or more event dissemination trees. Event dissemination trees are constructed by routing keys to event brokers in the overlay network. Every event dissemination tree has at least one special event broker functioning as a rendezvous node, described in the next section.

Rendezvous Nodes

In HERMES, a *rendezvous node* is used to ensure that all interested event brokers agree on the same set of event dissemination trees for a particular event type. When constructing a tree, event advertisements and subscriptions are routed towards the rendezvous node using the peer-to-peer routing substrate, so that, in the worst case, they join at the rendezvous node. The rendezvous node must exist at a globally known location in the network. The idea of rendezvous nodes was first introduced in the context of *core-based trees* [BFC93] for building multicast trees. However, the content-based routing algorithm of HERMES does not require that all event publications are routed through the rendezvous node, thus avoiding a potential bottleneck at that node.

The type-based routing layer of HERMES maintains at least one rendezvous node for every event type in the system because separate event dissemination trees per event publisher are built for each type. Any event broker in HERMES can assume the role of a rendezvous node for one or more event types. A rendezvous node is automatically created when a new event type is added to HERMES. Once an event broker has become a rendezvous node, it is responsible for managing that particular event type. The event broker acting as the rendezvous node is chosen by hashing the unique event type name as a key to a destination event broker in the distributed hash table. Due to the properties of the distributed hash table, the chosen event broker will be globally agreed upon by all brokers so that every broker can use the peer-to-peer routing substrate to send messages to this rendezvous node.

A rendezvous node manages the authoritative version of the event type schema used for type-checking events of that type. When a rendezvous node is set up, it stores the event type schema for the new type in its event type repository. It also supports a query interface to browse for an event type schema and a management interface to modify it. To reduce the load on the rendezvous node, copies of the event type schema are cached at other event brokers for faster type-checking of event subscriptions and publications. Because an event type may be a descendant from a parent type, a rendezvous node can be associated with a parent rendezvous node. This may make it necessary to traverse a chain of rendezvous nodes to collect all type information in the inheritance hierarchy when type-checking a child event type. If an event client attempts to register an already existing event type, the rendezvous node can reject the type message and notify the client of the name conflict.

Rendezvous nodes do not contain any state about the event dissemination trees itself, which makes them simple to replace in case of failure. When a rendezvous node fails, a new rendezvous node will take over because of the adaptation of the peer-to-peer routing substrate. To prevent the event type schema information from being lost, it is usually replicated across multiple nodes. We will describe several strategies for installing redundant rendezvous nodes, when discussing Hermes's fault-tolerance mechanisms in Section 4.3.7.

4.2.2 Event Clients

The event clients in Hermes follow the component model for an event-based middleware. They are light-weight, language-dependent components without middleware functionality that connect to a local event broker in order to request middleware services. Hermes has event publishers and event subscribers as clients. For usability, an event client locally exports a language-dependent event client API to the application and API calls are then passed on to the language-independent client interface of Hermes event brokers. The communication with the client interface of the event broker is XML-based and hidden from the application.

As can be seen from the event publisher and subscriber APIs exported to applications in Tables 4.2 and 4.3, every API method includes the client identity (in `pub` or `sub`) and a set of credentials (in `c`) for an access control decision by the local event broker. Both types of event clients support methods to connect to a Hermes event broker, which then becomes a publisher-hosting or subscriber-hosting broker. To increase reliability and availability, an event client may maintain connections to several local event brokers. The `getLocalBrokers` method returns the set of local event brokers and a disconnect method removes the client from a broker. In addition, both client APIs contain specific methods that depend on the capacity of the event client as an event publisher or subscriber.

Event Publishers

The interface exported by an event publisher, as listed in Table 4.2, is primarily used to publish events in Hermes. In addition to handling connections to local event brokers, the interface also has methods for event type management. When an event publisher adds a new event type to Hermes with the `addEventType` method, it becomes the *event type owner* for this type. Only the event type owner is allowed to modify or remove the event type from the system. Any publisher, pending the possession of the required credentials, can query an event type schema by calling `getEventType`.

Before an event publisher can publish an event, it has to `advertise` the event type, which will trigger an event advertisement message that updates the event dissemination tree. A corresponding `unadvertise` method undoes a previous advertisement. Three methods exist to publish an event. A call to `publishType` publishes an event that will only match type-based subscriptions that are of the correct event type (or parent event type), whereas `publishTypeAttr` only matches type- and attribute-based subscriptions that have a matching content-based filter on the event attributes. The third method, `publish`, publishes an event that will potentially match both types of subscriptions. Note that in Section 4.3.6 we will show that the `publish` method

Returns	API Call	Parameters
void	*connectPublisher*	(Publisher pub, Credentials c, EventBroker broker)
BrokerSet	*getLocalBrokers*	(Publisher pub, Credentials c)
void	*disconnectPublisher*	(Publisher pub, Credentials c, EventBroker broker)
void	*addEventType*	(TypeOwner owner, Credentials c, TypeSchema schema)
void	*modifyEventType*	(TypeOwner owner, Credentials c, TypeSchema schema)
TypeSchema	*getEventType*	(Publisher pub, Credentials c, EventType type)
void	*removeEventType*	(TypeOwner owner, Credentials c, EventType type)
void	*advertise*	(Publisher pub, Credentials c, EventType type)
void	*unadvertise*	(Publisher pub, Credentials c, EventType type)
void	*publish*	(Publisher pub, Credentials c, Event event)
void	*publishType*	(Publisher pub, Credentials c, Event event)
void	*publishTypeAttr*	(Publisher pub, Credentials c, Event event)

Table 4.2: The Hermes event publisher API

is more efficient than merely calling `publishType` and `publishTypeAttr` in sequence because event publications are less likely to be duplicated during routing.

Event Subscribers

Event subscribers have an interface that enables them to subscribe to events. The `getEventType` method in Table 4.3 returns the event type schema so that a type-safe subscription can be made. Analogous to the API of event publishers, the subscribe and unsubscribe methods come in three flavours. A subscription according to type-based routing is created by `subscribeType`, `subscribeTypeAttr` submits a type- and attribute-based subscription with a content-based `filter` expression, and `subscribe` will match events following either routing algorithm.

All three subscription methods include a `callback` parameter for the asynchronous notification of event publications. The callback interface consists of a single `notify` method, shown in Table 4.4. It is invoked by the event subscriber to deliver a matching event publication from the local event broker to the application.

Returns	API Call	Parameters
void	*connectSubscriber*	(Subscriber sub, Credentials c, EventBroker broker)
BrokerSet	*getLocalBrokers*	(Subscriber sub, Credentials c)
void	*disconnectSubscriber*	(Subscriber sub, Credentials c, EventBroker broker)
TypeSchema	*getEventType*	(Subscriber sub, Credentials c, EventType type)
void	*subscribe*	(Subscriber sub, Credentials c, EventType type, Filter filter, Callback callback)
void	*subscribeType*	(Subscriber sub, Credentials c, EventType type, Callback callback)
void	*subscribeTypeAttr*	(Subscriber sub, Credentials c, EventType type, Filter filter, Callback callback)
void	*unsubscribe*	(Subscriber sub, Credentials c, EventType type, Callback callback)
void	*unsubscribeType*	(Subscriber sub, Credentials c, EventType type, Callback callback)
void	*unsubscribeTypeAttr*	(Subscriber sub, Credentials c, EventType type, Filter filter, Callback callback)

Table 4.3: The HERMES event subscriber API

4.3 Routing Algorithms

In this section the two routing algorithms for event dissemination in HERMES, *type-based routing* and *type- and attribute-based routing*, are described. They each have their own trade-offs, which is why event clients can choose the algorithm to use depending on their subscription or publication patterns. Type-based routing combines advertisements and subscriptions with core-based trees [BFC93] over a distributed hash table, where the core is the event broker that functions as the rendezvous node. Special consideration has to be given to event typing and type inheritance. The more advanced type- and attribute-based routing algorithm extends core-based trees with the technique of *reverse path forwarding* [DM78], which is used in publish/subscribe systems to install filtering state at event brokers.

We start the presentation of the two routing algorithms with an overview of the message types in HERMES. After that, the main data structures maintained at event brokers participating in routing are explained in Section 4.3.2. A description of the type-based routing algorithm is given

Returns	API Call	Parameters
void	*notify*	(Event event)

Table 4.4: The HERMES event subscriber callback API

in Section 4.3.3, with the necessary changes to support event type inheritance detailed in the following section. In Section 4.3.5 we introduce type- and attribute-based routing. Issues related to the execution of both algorithms in a single Hermes deployment and a formalisation of their behaviour are given in Section 4.3.6. We finish with a survey of fault-tolerance mechanisms supported by the algorithms in Section 4.3.7.

4.3.1 Message Types

To disseminate events in the event-based middleware, Hermes event brokers exchange four types of messages, namely *type*, *advertisement*, *subscription*, and *publication* messages. The latter three types correspond to the messages introduced in the event and routing models in Sections 3.3.1 and 3.3.3. Type messages are specific to the management of rendezvous nodes and the type-checking of event data. Both routing algorithms use the same set of messages except that subscription and publication messages have an extra field to disambiguate whether they are part of type-based or type- and attribute-based routing. Subscription messages in type- and attribute-based routing also include a content-based filter expression. All message types contain source and destination address fields, where the source address is the address of the most recent event broker to process that message and the destination address is a destination in the peer-to-peer routing substrate.

In addition to these four types of messages, there are unadvertisement and unsubscription messages, which are inverses of the corresponding messages described above. The routing algorithms use them to remove state from event brokers, but for all practical purposes they behave in the same manner as their positive counterparts. Next we will explain the purpose and structure of each message type in turn.

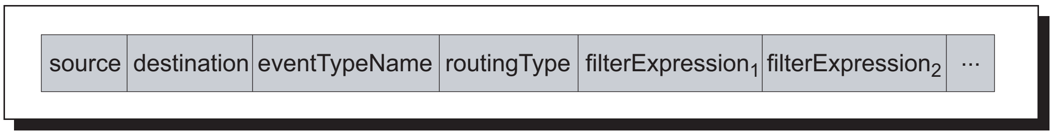
Type Messages. *Type messages* originate at publisher-hosting brokers when an event publisher tries to add a new event type to the system. They are used to set up the rendezvous node for a new event type and add its schema to the type repository maintained at the rendezvous node. In addition to the `eventTypeName` and the `eventTypeSchema` fields, an `operation` field determines whether to add a new event type, or modify or remove an existing one. The `destination` address is always the event broker that acts as the rendezvous node for the event type. Type messages are the same for type-based and type- and attribute-based routing.

source	destination	operation	eventTypeName	eventTypeSchema

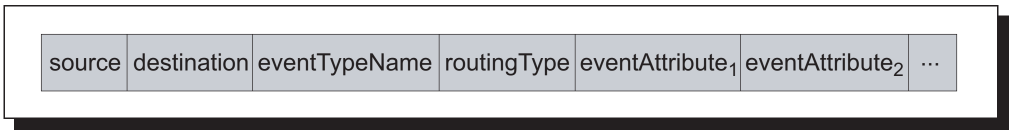
Advertisement Messages. An event publisher that is willing to publish events of a given event type causes its hosting event broker to send an *advertisement message*. Advertisement messages are routed towards the rendezvous node and contain the `eventTypeName` of the type to be published. At each event broker that an advertisement message passes through, it may create state in an advertisement routing table, which is later used to create an event dissemination tree.

source	destination	eventTypeName

Subscription Messages. A *subscription message* is sent by a subscriber-hosting broker when an event subscriber submits a subscription. Like advertisement messages, subscription messages are routed towards the rendezvous node. They contain an `eventTypeName` field and optional `filterExpression` fields if used with type- and attribute-based subscriptions, as indicated by the `routingType` field. Subscription messages may cause new entries to be added to subscription routing tables at event brokers.



Publication Messages. The fourth type of message is a *publication message* with a published event. It is sent by a publisher-hosting broker that received an event publication from an event publisher. The routing of publication messages is controlled by the advertisement and subscription routing tables at event brokers. These messages contain an `eventTypeName` field, a routing algorithm specification in a `routingType` field, and values for the event attributes in the `eventAttribute` fields.



4.3.2 Data Structures

The content-based routing algorithms in Hermes depend on two data structures maintained by all event brokers. An *advertisement routing table* records information about advertisements and a *subscription routing table* does the same for subscriptions. Both routing tables have an identical form and are thus instances of the same data structure. Their purpose is to keep track of the paths taken by advertisement/subscription messages that have passed through an event broker. This information is then sufficient to create event dissemination trees for the subsequent routing of event publication messages, coming from any number of event publishers.

For every advertisement/subscription, the routing table at an event broker contains an entry that associates it with two sets: the set of event brokers that *sent* the message to the current event broker, and the set of event brokers that *received* the message from the current event broker. The event brokers in these two sets must be neighbouring event brokers because only these event brokers can directly communicate with the current event broker. An event broker is only added to one set if the message originated or terminated at the current event broker. Moreover, the routing table also allows efficient access to the coverage relation between the advertisements/subscriptions it stores, as defined in Section 3.3.1.

Definition 4.1 (Advertisement/Subscription Routing Table) *An* advertisement/subscription routing table $\mathrm{RT}_{\mathrm{adv/sub}}$ *contains a set of routing table entries,* RT_{E}*,*

$$\mathrm{RT}_{\mathrm{E}} \in \mathrm{RT}_{\mathrm{adv/sub}}.$$

An advertisement/subscription routing table entry RT_{E} *is a tuple,*

$$\mathrm{RT}_{\mathrm{E}} = (\mathrm{adv/sub},\ B_{\mathrm{from}},\ B_{\mathrm{to}}),$$

where adv/sub *is an advertisement/subscription,* B_{from} *is a set of event brokers that sent, and* B_{to} *is a set that received this advertisement/subscription.*

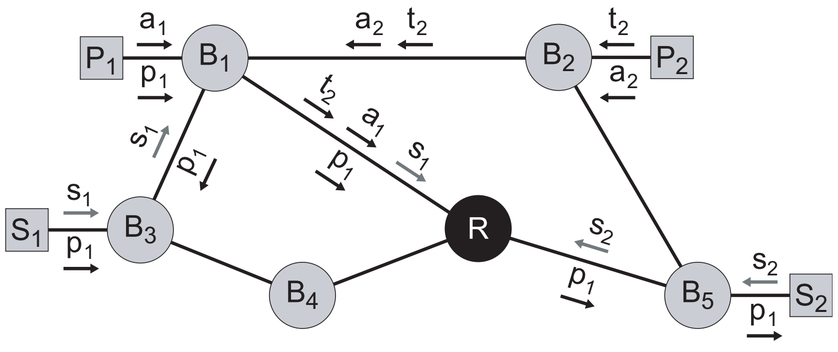


Figure 4.3: Type-based routing in HERMES

4.3.3 Type-Based Routing

As mentioned previously, the type-based routing algorithm disseminates event publications depending on the event type only. Before an event of a given event type can be published, the corresponding rendezvous node must be set up. This is done by routing a type message to the event broker that is the destination node for the hashed event type name in the distributed hash table. This event broker then becomes the rendezvous node for the new event type and stores the event type schema in its type repository. When one of the remaining three types of messages reaches an event broker, it is then processed as follows:

Advertisement Messages. An advertisement message is routed towards the rendezvous node for the specified event type. At every broker the advertisement is added to the advertisement routing table, recording the advertisement, the last, and the next event broker on the message path. However, if the next event broker on the path to the rendezvous node has already received a previous advertisement that covers the new one, the message is not forwarded but discarded. Once the advertisement message reaches the rendezvous node, it is added to the advertisement routing table and then also discarded.

Subscription Messages. A subscription message behaves analogously to an advertisement message. It is routed towards the rendezvous node and creates entries in subscription routing tables at each broker along the way. The message is discarded if the next broker has already received a previous, covering subscription. The subscription message is also discarded when it reaches the rendezvous node.

Publication Messages. Publication messages follow the forward path of matching advertisements, as stored in advertisement routing tables, up to the rendezvous node. Whenever they encounter an event broker with matching subscriptions, they follow the reverse paths from the subscription routing table. Note that because of best-effort semantics no state needs to be created at the event brokers that process publication messages, and they are never forwarded to an event broker that was the previous hop on the message path.

By following the type-based routing algorithm, an event publication is disseminated to all interested event subscribers that have submitted a matching subscription. The algorithm thus creates several event dissemination trees — one for each publisher-hosting broker— that are then used for the flow of event publications. In Figure 4.3 we illustrate the operation of type-based routing in HERMES with an example. There are six event brokers $B_{1...5}$ and R, two event subscribers $S_{1,2}$, and two event publishers $P_{1,2}$ in this deployment. The rendezvous node for the only event type in the system is at event broker R.

First the rendezvous node at event broker R is set up with the type message t_2 coming from event broker B_2 that is hosting event publisher P_2 with a request for event type creation. Now P_1 and P_2 advertise so that advertisement messages a_1 and a_2 are routed to the rendezvous node R. During this process, state in advertisement routing tables at event brokers B_1, B_2, and R is created. Since advertisement a_2 is covered by a_1, the message a_2 is discarded by event broker B_1. Next the two event subscribers S_1 and S_2 subscribe to this event type, triggering the routing of two subscription messages s_1 and s_2 to R. Again entries in the subscription routing tables at event brokers B_1, B_3, B_5, and R are established. As a result, two event dissemination trees have been created in the system. Note that no state is needed at event broker B_4 because it does not lie on one of the two trees.

When event publisher P_1 publishes an event, an event publication message p_1 is generated at event broker B_1. The state in the advertisement and subscription routing tables at event broker B_1 is given in Table 4.5. At the first hop, the publication message is sent to event broker R following the forward path of the advertisement message a_1, and to B_3 on the reverse path of the subscription message s_1. The event broker R forwards the message to B_5 so that it eventually reaches both event subscribers S_1 and S_2.

In type-based routing an event publication must reach the rendezvous node but may be discarded there because it has already encountered all matching event subscriptions on its path to the rendezvous node. Type- and attribute-based routing relaxes this requirement by not necessarily routing event publications through the rendezvous node and thus preventing it from becoming a bottleneck. In the next section we present a modified handling of advertisement and subscription messages in type-based routing so that event type inheritance with supertype subscriptions, as introduced in the event model in Section 3.3.1, is supported.

4.3.4 Type-Based Routing with Inheritance

Previously we argued that supertype subscriptions helped integrate object-oriented type inheritance with a typed event model. With supertype subscriptions, an event subscriber submitting a subscription for an event type will also be notified of published events of any of the descendant event types of the original subscription type. The simplest way to realise supertype subscriptions is to leave the responsibility of subscribing to all descendant event types with the event subscriber. However, this adds unnecessary complexity to event subscribers, which can be avoided by extending the type-based routing algorithm in HERMES to accommodate supertype subscriptions explicitly.

The type message that establishes a rendezvous node in type-based routing contains the schema of the new event type. This schema also includes the name of the parent event type. Therefore a rendezvous node can contact its parent rendezvous node to notify it that a new child event type has been added to the system. Since rendezvous nodes are now aware of all their descen-

Advertisement	$\mathbf{B_{from}}$	$\mathbf{B_{to}}$	**Subscription**	$\mathbf{B_{from}}$	$\mathbf{B_{to}}$
$\{a_1\}$	$\{\}$	$\{R\}$	$\{s_1\}$	$\{B_3\}$	$\{R\}$
$\{a_2\}$	$\{B_2\}$	$\{\}$			

Table 4.5: Routing tables at event broker B_1 using type-based routing

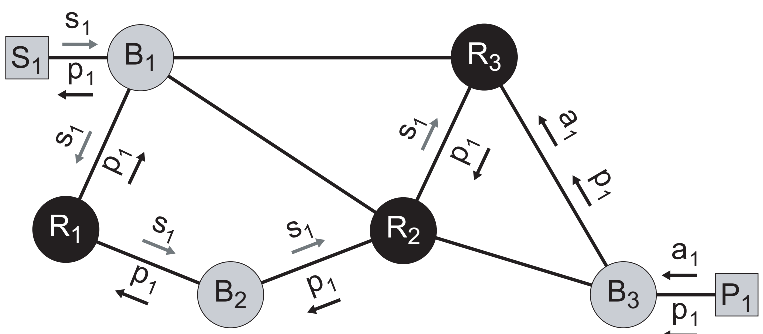


Figure 4.4: Supertype subscriptions with subscription inheritance in Hermes

dant types, supertype subscriptions can be supported by either advertisement or subscription inheritance routing.

Advertisement Inheritance Routing. When an advertisement message reaches a rendezvous node, the message continues propagation recursively to the rendezvous nodes of all ancestor event types. This results in additional state in advertisement routing tables. The processing of subscription messages is unchanged, meaning that they are only sent to the rendezvous node of the event type stated in the subscription. Now event publications, which follow the forward path of advertisements, will reach the rendezvous nodes of all ancestor event types and thus encounter all matching event subscriptions in the type hierarchy.

Subscription Inheritance Routing. With subscription inheritance routing, the roles of advertisement and subscription messages are interchanged. Advertisement messages are only sent to the rendezvous node for the event type from the advertisement. However, a subscription message that arrives at a rendezvous node continues to be routed to all rendezvous nodes of descendant types. As a consequence, a publication message, which follows the forward advertisement path to a rendezvous node of a descendant type, has to encounter all matching subscriptions for any ancestor event type.

In Figure 4.4 an example of subscription inheritance routing in Hermes is given using an event type hierarchy with three event types $\tau_{1,2,3}$ and corresponding rendezvous nodes $R_{1,2,3}$. The subtyping relation $\tau_3 \preceq \tau_2 \preceq \tau_1$ holds between the event types. Note how a subscription $s_1 : \tau_1$ at event broker B_1 creates extra state in subscription routing tables at event brokers B_2, R_2, and R_3 because of the additional routing of subscription messages due to inheritance. The event publication $p_1 : \tau_3$ is delivered to the event subscriber that submitted the supertype subscription s_1 since it is matched at rendezvous node R_3.

Whether to use advertisement or subscription inheritance routing depends on the ratio between advertisements and subscriptions. If event advertisements triggered by event publishers are more common, subscription inheritance routing has the advantage of a smaller overhead since advertisements are only routed to a single rendezvous node in the event type hierarchy. Conversely, if subscriptions are more common, it is more efficient to route event advertisement more widely in the system. We will revisit a similar strategy for dealing with multiple rendezvous nodes in Section 4.3.7 when looking at fault-tolerance mechanisms for content-based routing in Hermes.

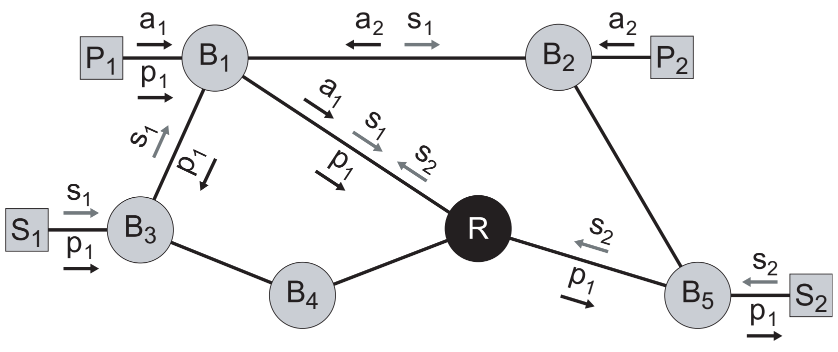


Figure 4.5: Type- and attribute-based routing in HERMES

4.3.5 Type- and Attribute-Based Routing

Type- and attribute-based routing is an extension of type-based routing that supports the content-based filtering of events according to filter expressions on the event attributes in event publications. It differs from type-based routing in that filtering state from subscriptions is installed close to publishers so that unmatched events can be discarded early during event dissemination. The same mechanism as for type-based routing is used to set up rendezvous nodes with type messages, and the processing of the remaining three message types is as follows:

Advertisement Messages. Advertisement messages are used in the same way as in the type-based routing algorithm. They are routed towards the rendezvous node for the advertised event type and create new entries in the advertisement routing tables of the event brokers that they pass through. The coverage relation between advertisements suppresses the forwarding of more specific advertisements that are covered by previous, less specific ones. An advertisement message is not forwarded further than the rendezvous node.

Subscription Messages. A subscription message is also routed towards the rendezvous node for its event type. However, the subscription message also follows the reverse path of any covering advertisements along the way. At each hop an entry in the subscription routing table is created and the message may not be further propagated because of subscription coverage. The regular routing of a subscription message finishes at the rendezvous node.

Publication Messages. The routing of publication messages is only controlled by the state in subscription routing tables. Publication messages follow the reverse path of matching event subscriptions. This means that they are not routed towards the rendezvous node for their event type. As they are matched against type- and attribute-based event subscriptions with attribute predicates, event publications are progressively filtered according to event subscribers' interests.

Advertisement	**B_{from}**	**B_{to}**	**Subscription**	**B_{from}**	**B_{to}**
$\{a_1\}$	$\{\}$	$\{R\}$	$\{s_1\}$	$\{B_3\}$	$\{R, B_2\}$
$\{a_2\}$	$\{B_2\}$	$\{\}$	$\{s_2\}$	$\{R\}$	$\{\}$

Table 4.6: Routing tables at event broker B_1 using type- and attribute-based routing

We illustrate type- and attribute-based routing in Hermes with the example in Figure 4.5. The setup of event brokers and event clients is the same as in Figure 4.3. First, the two event publishers P_1 and P_2 advertise the only event type in the system and thus cause the sending of advertisement messages a_1 and a_2. These two messages add entries to the advertisement routing tables at event brokers B_1, B_2, and R. When event subscriber S_1 decides to subscribe, the subscription message s_1 is routed towards the rendezvous node R. The routing table state at event broker B_1 is summarised in Table 4.6. There it encounters a matching event advertisement a_2 and is therefore forwarded to event broker B_2. It is also delivered to event broker R because of rendezvous node routing. The subscription s_2 from subscriber S_2 is routed similarly, except that it is discarded by event broker B_1 due to coverage with subscription s_1. Finally the event publication p_1 is routed along the reverse path of subscription s_2, namely via B_1, R, and B_5. It also follows the reverse path of subscription s_1 to event broker B_3 so that it is successfully delivered to event subscribers S_1 and S_2.

When compared to type-based routing, type- and attribute-based routing creates more state in subscription routing tables in the system. This is because event publications are filtered as close to event publishers as possible, requiring more filtering state in the system. An advantage of this routing scheme is that event publications do not have to reach the rendezvous node any more as they are not routed along the path taken by event advertisements. Hence the rendezvous node cannot become a bottleneck in event dissemination. In practice, accepting more subscription state to make routing of event publications more efficient is a good trade-off because publications are more frequent than subscriptions in most real-world applications since event subscribers typically stay subscribed for many publications. A more in-depth evaluation of the Hermes routing algorithms will be provided in Chapter 5.

4.3.6 Combined Routing

For maximum flexibility, a Hermes deployment should be able to support both type-based and type- and attribute-based routing. This enables event clients to choose the service that best suits their needs. If an event client does not need content-based filtering, a type-based subscription has the advantage of creating less subscription state at event brokers. Type- and attribute-based routing is necessary if event clients rely on more expressive content-based subscriptions with filter expressions. Ideally an event publisher prefers to only publish a single event publication, which will then be delivered to all type-based and type- and attribute-based subscribers.

The combination of type-based and type- and attribute-based routing in a single system involves some subtleties due to interference between the two algorithms. In a naïve approach, where the same advertisement, subscription, and publication messages are processed by both algorithms, events might be delivered to event subscribers multiple times. This can occur when an event publication encounters matching type-based event advertisements on the reverse path of a type- and attribute-based subscription.

An extreme solution would be to completely separate type-based and type- and attribute-based routing, each having its own set of messages. Although this would solve the problem, it would lead to a less efficient system in terms of the number of messages sent. For example, an event publication that is to be matched against both type-based and type- and attribute-based subscriptions would have to be sent twice. Instead, we propose a mechanism in which event publications record whether they are following a type-based or an type- and attribute-based event dissemination tree. A new decision about the affiliation of an event publication is made at every event broker in case the two kinds of event dissemination trees overlap.

```
processAdvertisementMsg(adv):
advRT(adv).From $\leftarrow$ advRT(adv).From $\cup$ adv.LastNode
IF (adv.NextNode $\notin$ covered(adv,advRT).To) THEN
  advRT(adv).To $\leftarrow$ advRT(adv).To $\cup$ adv.NextNode
ELSE
  adv.NextNode $\leftarrow$ NULL
$\forall$sub IN covered(adv,subRT)
  IF (sub.IsTypeAttrBased) THEN
    processSubscriptionMsgTypeAttrBased(sub)

processSubscriptionMsgTypeBased(sub):
subRT(sub).From $\leftarrow$ subRT(sub).From $\cup$ sub.LastNode
IF (sub.NextNode $\notin$ covered(sub,subRT).To) THEN
  subRT(sub).To $\leftarrow$ subRT(sub).To $\cup$ sub.NextNode
ELSE
  sub.NextNode $\leftarrow$ NULL

processSubscriptionMsgTypeAttrBased(sub):
processSubscriptionMsgTypeBased(sub)
$\forall$node IN covered(sub,advRT).From
  IF (node $\notin$ covered(sub,subRT).To) THEN
    subRT(sub).To $\leftarrow$ subRT(sub).To $\cup$ node
    send(sub,node)

processPublicationMsg(event):
nodeSetTypeBased $\leftarrow$ covered(event,advRT).To
    $\cup$ covered(event,subRT).From
nodeSetTypeAttrBased $\leftarrow$ covered(event,subRT).From
IF (event.IsTypeBased) THEN
  $\forall$node IN nodeSetTypeBased
    eventTypeBased $\leftarrow$ event
    eventTypeBased.IsTypeAttr $\leftarrow$ (node $\in$ nodeSetTypeAttrBased)
    nodeSetTypeAttrBased $\leftarrow$ nodeSetTypeAttrBased $\setminus$ node
    send(eventTypeBased,node)
IF (event.IsTypeAttrBased) THEN
  event.IsTypeBased $\leftarrow$ FALSE
  $\forall$node IN nodeSetTypeAttrBased
    send(event,node)
```

Figure 4.6: Pseudo code for combined event routing in Hermes

We define the complete routing algorithm of Hermes, which combines type-based and type- and attribute-based routing, with the pseudo code in Figure 4.6. The routing algorithm is divided into four functions that are called when the specified message type is received at an event broker. For simplicity of presentation, we assume that any necessary rendezvous nodes are set up and supertype subscriptions are not supported.

At each hop, an advertisement that is routed towards the rendezvous node is processed by the `processAdvertisementMsg` method in line 1. A new entry in the advertisement routing table (advRT) is created that records the advertisement and the event broker it came from (line 2). If the advertisement is not covered by an earlier one (line 3), it continues to be forwarded to the next event broker on the path and this is recorded in the advRT (line 4), otherwise the advertisement is discarded (line 6). The advertisement may trigger the reverse propagation of a previous type- and attribute-based subscription, so that in line 7 it is matched against existing subscriptions in the subscription routing table (subRT). Every matched type- and attribute-based subscription (line 8) is possibly further propagated by a call to the `processSubscriptionMsg-TypeAttrBased` method (line 9).

A type-based subscription is processed by the `processSubscriptionMsgTypeBased` method in lines 11–16. It is analogous to the first part of advertisement processing described above, except that entries are added to the subRT instead of the advRT. The `processSubscriptionMsgType-AttrBased` method is responsible for type- and attribute-based subscriptions (line 18). They are first processed like type-based subscriptions (line 19), propagating towards the rendezvous node, but then additional filtering state needs to be set up closer to event publishers (lines 20–23). Therefore the subscription follows the reverse path of all matching advertisements (line 20), unless it is already covered by a previous subscription (line 21). A new entry in the subRT is created (line 22) when the subscription is sent to a node (line 23).

Finally, event publications are handled in the `processPublicationMsg` method (line 25). They need to be routed differently depending on whether they follow only type-based, only type- and attribute-based, or both types of subscriptions. A publication message has hence two flags (`IsTypeBased` and `IsTypeAttrBased`) to indicate its routing semantics. At first the next hop sets for type-based (line 26) and type- and attribute-based routing (line 28) are computed. If the publication came along a type-based event dissemination tree (line 29), it continues to be routed this way (lines 30–34), but it is removed from the type- and attribute-based destination set (line 33) so that it is not transmitted again to the same next broker on a type- and attribute-based event dissemination tree. Type- and attribute-based dissemination is dealt with in line 35 and the event publication is disseminated to the remaining next hops (lines 37–38).

4.3.7 Fault Tolerance

The reliability model for an event-based middleware states that event dissemination should be robust against failure. Both routing algorithms used by Hermes have built-in fault-tolerance features that enable event brokers to recover to a consistent system state after failure. In this section we describe how Hermes handles network and middleware failure, distinguishing between both classes of failure.

Failure in the underlying physical network results in the inability of event brokers to communicate with each other. Since all communication in the type-based and type- and attribute-based routing layers is done through the primitives provided by the overlay routing layer, we assume that after a network failure the overlay network will adapt and connectivity between event brokers will eventually be restored. Since this may cause a change in the routing path

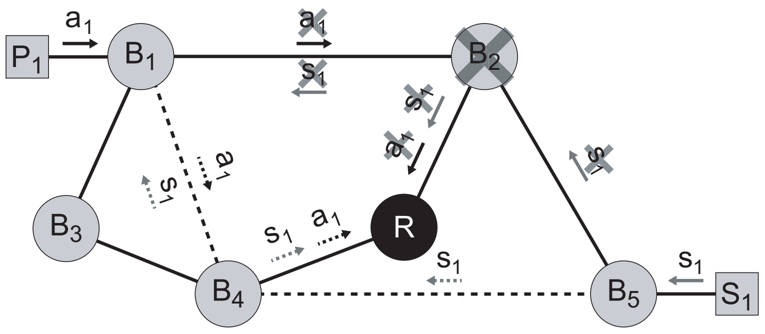


Figure 4.7: Fault-tolerance using type- and attribute-based routing in Hermes

of advertisement and subscription messages, care must be taken to restore advertisement and subscription routing tables to a consistent state.

A mechanism for repairing routing tables is also required to handle the failure of a middleware component, such as an event broker. In Hermes, event brokers periodically exchange *heartbeat messages* with their neighbouring event brokers in order to detect failure. All entries in routing tables consist of *soft state* and are associated with leases. An entry will automatically be removed from the routing table, unless its lease is renewed periodically by a heartbeat message from the event broker that caused the entry to be added, by sending the original advertisement or subscription message.

When an event broker fails, the subscription and advertisement messages that it forwarded to other event brokers expire and are removed from the routing tables. In order to repair such a damaged event dissemination tree, the event broker that detected the failure of a neighbouring event broker resends all advertisement and subscription messages that previously went via the failed broker to the rendezvous node. The peer-to-peer routing substrate ensures that the messages reach the rendezvous node via a different path. New state is established at event brokers that were previously not part of the event dissemination tree. Due to properties of the peer-to-peer routing substrate, it is likely that a new route quickly joins an old path to a rendezvous node so that existing state in routing tables will be reused as much as possible [CDHR02].

The failure of an event broker in Hermes is demonstrated with an example in Figure 4.7. The event broker B_1 detects the failure of event broker B_2 because of an absence of heartbeat messages. As a result, it resends advertisement a_1 to the rendezvous node R, which was previously propagated by event broker B_2. Once the overlay routing layer has adapted to the failure of node B_2, it changes the set of neighbouring event brokers at broker B_1 and forms a new connection through broker B_4. The advertisement message is routed via B_4, and new state is created in the advertisement routing table. The same behaviour is repeated for the subscription s_1 stored at event broker B_5. The only special case that remains to be handled is the failure of the rendezvous node itself.

Rendezvous Node Failure

Event routing in Hermes does not work without the existence of at least one rendezvous node for an event type. In addition, the authoritative version of the event type schema for type-checking is stored at the rendezvous node. To prevent rendezvous nodes from being single points of failure, they are replicated so that another replica can take over when an event broker hosting a rendezvous node fails.

Replication can be handled by the overlay routing layer because it is often directly supported by a distributed hash table. For example, the destination node for a rendezvous node replica can be obtained by concatenating a *salt value* to the event type name before computing the hash function [ZKJ01]. If an event broker or client does not receive a response from the primary rendezvous node, it attempts to contact one of the replicas with its salt value. However, a better technique is to make access to replicas transparent to higher event routing layers. To do this in HERMES, we rely on a property of the peer-to-peer routing substrate that guarantees that a key in the distributed hash table is always routed to the numerically closest, live node in the system. This guarantee is implemented using Pastry's leaf set mechanism, described in Section 2.3.1. Therefore, every rendezvous node is replicated across all nodes in its leaf set so that messages are transparently routed to a live rendezvous node replica.

Another important aspect of replication is maintaining consistency between replicas. This involves synchronising event type repositories and keeping advertisement and subscription routing state consistent. Since an event-based middleware is a large-scale, distributed system, a protocol for maintaining weakly consistent state between event type repositories is most suitable. The synchronisation of routing state is more subtle because inconsistent state at any time may result in incorrect event routing. The most robust solution is to route every advertisement and subscription message to all replicas in the system but this increases the number of messages sent. We adopt a weaker, more efficient solution, in which a single event dissemination tree is constructed that includes all rendezvous node replicas. Even though this means that some advertisement and subscription state may be lost when one rendezvous node replica fails, this can be handled by the fault-tolerance mechanism for repairing event dissemination trees explained in the previous section. Our proposed solution is similar to multicast with multiple cores [ZF01].

We identify four different approaches for exchanging advertisement and subscription messages between rendezvous node replicas in order to unify them into a single event dissemination tree with redundant rendezvous nodes.

Redundant Client Advertisements. When an event publisher causes a publisher-hosting broker to route an advertisement message to a rendezvous node, separate advertisement messages are sent to all rendezvous node replicas. Event subscriptions may then be addressed to a single live rendezvous node replica since they are guaranteed to encounter all matching advertisements.

Redundant Client Subscriptions. With redundant client subscriptions, the strategy is reversed. Event subscriptions are issued to all rendezvous node replicas by a subscriber-hosting broker and publisher-hosting brokers advertise to only one rendezvous node replica.

Redundant Rendezvous Node Advertisements. Instead of the publisher-hosting broker, it becomes the responsibility of a rendezvous node replica to contact the other replicas. A publisher-hosting broker sends a single advertisement message to any rendezvous node replica, but the replica then forwards the advertisement message to all other replicas. Event subscribers may contact any replica for their subscriptions.

Redundant Rendezvous Node Subscriptions. Finally, event subscriptions can be sent to a single rendezvous node replica, which then subscribes to all other replicas on the event subscriber's behalf. Event publishers only advertise to one replica.

In Figure 4.8 rendezvous node replication with redundant advertisements is shown. The rendezvous node for the only event type in the system has three replicas, $R_{1,2,3}$. When the event advertisement a_1 reaches the rendezvous node replica R_1, it is forwarded to R_2 and R_3. This sets up further state in the advertisement routing tables at event brokers B_2, R_2, and R_3. When

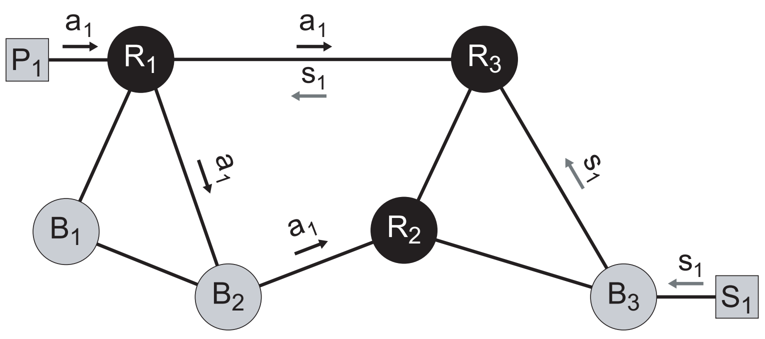


Figure 4.8: Rendezvous node replication with redundant advertisements in Hermes

the type- and attribute-based subscription s_1 reaches rendezvous node replica R_3, it encounters an entry for advertisement a_1 in the advertisement routing table and follows its reverse path to the original replica R_1.

The four approaches for integrating rendezvous node replicas with event routing in Hermes have different trade-offs. As was the case for the inheritance routing schemes in Section 4.3.4, the decision whether to use redundant advertisement or subscription routing depends on the ratio of advertisement and subscription messages in the system. If subscriptions are more common than advertisements, it is better to add more state to advertisement routing tables using redundant advertisement messages. Introducing redundant messages at the client-hosting brokers has the benefit that event brokers can choose the best replica in terms of proximity or load for local event dissemination. The downside is that event brokers have to be explicitly aware of replication, which means that transparent routing to replicas without salt values cannot be applied. In contrast, if redundant messages are sent by rendezvous nodes, rendezvous node replication is hidden from client-hosting brokers.

4.4 Implementation

In this section we present our prototype implementation of Hermes. There exist two versions of Hermes: an implementation in our distributed systems simulator DSSim that will be used for the evaluation of the Hermes' event routing algorithms in Chapter 5 and a regular implementation that provides event broker and client components that can be deployed on the Internet. Both versions are implemented in Java and share most of the source code, in particular the implementation of the event routing algorithms is the same. The main difference is that the simulational prototype does not handle messaging issues, such as the serialisation of advertisements, subscriptions, and publications to XML messages and inter-broker communication.

In the following we focus on the regular version and look at the communication aspects of Hermes. Event brokers communicate by sending XML messages through TCP connections between event brokers. The use of the SOAP messaging protocol, as outlined in Section 2.1.3, would have been an equally valid choice and was investigated, but the SOAP specification was still in a preliminary stage when the development of Hermes began. The Hermes implementation relies on the rich type model of the XML Schema specification. The communication protocol for the client API between event brokers and clients is also based on XML messages. As a result, event clients have an XML-to-event binding layer that maps programming language objects to XML messages and vice versa.

Returns	API Call	Parameters
`NodeID`	*initPanNode*	`(PanCallback callback)`
`void`	*leavePanNode*	`()`
`void`	*panRouteMessage*	`(Message msg, NodeID node)`
`Message`	*panRouteRequestReplyMessage*	`(Message msg, NodeID node)`
`void`	*panSendMessage*	`(Message msg, Address addr)`
`Message`	*panSendRequestReplyMessage*	`(Message msg, Address addr)`

Table 4.7: The Pan API

We start with an overview of the implementation of Pan, our peer-to-peer routing substrate, in the next section. In Section 4.4.2 we show how the Hermes implementation handles event data by describing the format of XML messages and the XML-to-event binding used for programming language integration. Features concerning the extensibility of event brokers are discussed in Section 4.4.3.

4.4.1 Pan

The distributed hash table implementation for the overlay routing layer in the Hermes architecture is called Pan. It implements a peer-to-peer routing substrate that uses the Pastry routing algorithm [RD01]. The main parts of Pan are the routing protocol for messages and the join protocol to add new nodes to the network. We verified the correctness of the Pan implementation in our distributed systems simulator DSSim by successfully reproducing the average hop count and latency results from an evaluation of Pastry [CDHR02] within a small error.

Every node in Pan has a unique, randomly distributed, 128-bit nodeID grouped into digits with base 2^4. String values, such as event type names, are hashed with a SHA-1 hash function [FIP95] to convert them to nodeIDs for the routing of messages. The main data structures of Pan are a routing table and a leaf set, which were described in Section 2.3.1. The nodes in these two data structures taken together form the set of neighbouring event brokers for a broker. Pan also maintains a *neighbourhood set* that was part of early versions of the Pastry routing algorithm. A neighbourhood set keeps track of nodes that are physically close to a given node. In Hermes it is used by event brokers to inform clients of other brokers that could potentially become their local event brokers in case of failure.

The API methods of Pan that are called by the type-based and type- and attribute-based routing layers in Hermes are listed in Table 4.7. The join protocol to add a new node to the overlay routing network is invoked through the `initPanNode` method. It requires a reference to an implementation of the `callback` API described below and returns a randomly assigned nodeID for the new node. The method `leavePanNode` removes the node from the network. The main operation of the distributed hash table is invoked by calling `panRouteMessage`, which routes the message `msg` to the destination nodeID `node`. For communication with rendezvous nodes, the original Pastry interface was extended with a `panRouteRequestReplyMessage` method. It provides overlay communication with request/reply semantics so that the destination node for a message can directly return a response message to the sender. Finally, the last two methods allow direct node communication when the destination IP address is known, bypassing multi-hop routing in the overlay network.

Returns	API Call	Parameters
`void`	*deliverMessage*	`(NodeID thisNode, Message msg, NodeID sourceNode, NodeID lastNode, NodeID destNodeID)`
`Message`	*deliverRequestMessage*	`(NodeID thisNode, Message msg, NodeID sourceNode, NodeID lastNode, NodeID destNodeID)`
`NodeID`	*forwardMessage*	`(NodeID thisNode, Message message, NodeID sourceNode, NodeID lastNode, NodeID nextNode, NodeID destNodeID)`
`void`	*leafSetChanged*	`(NodeID thisNode, LeafSet leafSet)`

Table 4.8: The Pan callback API

Whenever a message is received at a Pan node, an invocation to the Pan callback interface, shown in Table 4.8, is performed. A call to `deliverMessage` indicates that the message `msg` has reached its final destination node, in other words, its key hashes to the current nodeID. Additional information in the callback are the nodeIDs of the original sender (`sourceNode`), of the last hop (`lastNode`), and of the original destination key (`destNodeID`). The `deliverRequestMessage` method is similar, except that it expects a reply message for the request message `msg` that will be returned to the sender. At every hop during message routing, the `forwardMessage` method is called. The event routing layers in Hermes use this callback to update advertisement and subscription routing tables. This method includes a `nextNode` parameter that holds the nodeID of the next hop on the routing path to the destination node. The routing of a message in transit can be influenced by altering this value. Since the leaf set of a rendezvous node is used to manage rendezvous node replicas transparently, changes to the leaf set have to be signalled with the `leafSetChanged` callback method so that replicated type repositories can be synchronised by the event-based middleware layer in Hermes.

4.4.2 Event Data

For communication purposes, the prototype implementation of Hermes must be able to express the event data as messages. As argued in Section 3.3.1, we decided to use the XML Schema specification to represent type, advertisement, subscription, and publication messages in Hermes in the form of XML Schema instance documents. Event types referred to by these messages are defined as XML Schema documents. This has the benefit that the type-checking of subscription and publication messages is automatically performed by a validating XML Schema parser, such as the Apache Xerces Parser [Apa01], which is used by Hermes. The XML Namespaces specification [W3C99a] prevents name conflicts between user-defined event types and other data.

Figure 4.9 gives an example of a `BaseEventType` defined in XML Schema. This type can be chosen as an ancestor for all event types in Hermes because it includes event attributes that are useful to most event subscribers. An event type definition in Hermes populates a separate XML namespace that is referenced in line 4. The event type schema is declared as a complex XML Schema type in line 6. It has three optional event attributes (lines 8–10) that are defined with their datatypes. An `id` field stores a publisher-specific identifier, the `publisher` field associates the event with an event publisher, and a timestamp is stored in the `timestamp` attribute. The actual schema element for the new event type is then defined in line 14.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:t="http://www.cl.cam.ac.uk/opera/hermes/type"
  targetNamespace="http://www.cl.cam.ac.uk/opera/hermes/type">

  <xsd:complexType name="BaseEventType">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:long" minOccurs="0"/>
      <xsd:element name="publisher" type="xsd:string" minOccurs="0"/>
      <xsd:element name="timestamp" type="xsd:dateTime" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="BaseEvent" type="t:BaseEventType"/>
</xsd:schema>
```

Figure 4.9: An event type schema defined in XML Schema

Event type definitions, such as this one, are stored in the event type repository maintained at the rendezvous node for the type. Subscriptions and publications can then be type-checked against the schema from this repository. Type-checking is usually only performed at client-hosting brokers when a subscription or publication message is received from an event client. To do type-checking, the event broker first attempts to obtain the corresponding event type schema from its local type repository cache. If this fails, it will contact the event type repository at the rendezvous node and retrieve the authoritative version of the schema. Next, the event data is validated against the schema by an XML parser and only admitted for further routing in Hermes if this succeeds. Note that event brokers need to be aware of inheritance relationships between event types in order to compute coverage relations between advertisements, subscriptions, and publications. The local cache of the event type repository at an event broker is kept up-to-date using *meta-events* published by the rendezvous node that indicate that an event type schema has evolved. They are disseminated using the event-based middleware with the event brokers acting as event subscribers.

Message Formats

The Hermes implementation has its own XML Schema definition for valid messages that can be exchanged between event brokers. An event broker that receives a message from another broker may check its conformance by validating it against the Hermes message schema and any other necessary event type schemas. Hermes messages reside in a private namespace. Using XML has the advantage that the messaging format can be extended with new elements as extra functionality is added to Hermes. In the following we give examples of type, advertisement, subscription and publication XML messages used by Hermes.

Type Message. A type message that sets up a rendezvous node for a new event type is given in Figure 4.10. The message identifies itself as a type message (line 3) and defines a new event type called `LocationEvent` that extends the previously defined `BaseEventType` in line 4. Lines 6–21 are the XML Schema definition of the new event type within the type message referring back to the parent type schema in line 10.

```
<?xml version="1.0" encoding="UTF-8"?>
<h:hermes xmlns:h="http://www.cl.cam.ac.uk/opera/hermes">
  <type>
    <addtype typename="LocationEvent" extends="BaseEvent">

      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:t="http://www.cl.cam.ac.uk/opera/hermes/type"
        targetNamespace="http://www.cl.cam.ac.uk/opera/hermes/type">

        <include schemaLocation="BaseEvent"/>

        <complexType name="LocationEventType">
          <complexContent>
            <extension base="t:BaseEventType">
              <all> <element name="position" type="xsd:string"/> </all>
            </extension>
          </complexContent>
        </complexType>

        <element name="LocationEvent" type="t:LocationEventType"/>
      </xsd:schema>

    </addtype>
  </type>
</h:hermes>
```

Figure 4.10: An XML definition of a type message

Advertisement Message. Advertisement messages are the simplest kind of message because they only need to include the event type that is being advertised. In Figure 4.11 the `advertise` element advertises the event type `LocationEvent` (line 4).

Subscription Message. A subscription message is first sent by a subscriber-hosting broker and contains a type-based or type- and attribute-based subscription. An example of a type- and attribute-based subscription with a filter expression is given in Figure 4.12. The message specifies the event type of interest (line 4) and a filter expression on the event attributes in lines 5–7, which filters on people with particular ids in an office. The filter expression is defined in the XPath language [W3C99b] in line 6, although any other filtering language could be used, as well. Unfortunately, XPath does not use XML syntax, but instead has its own textual representation for filter expressions over a single XML document. In HERMES, XPath filter expressions are matched by an XPath processor, in our case Apache Xalan [Apa03].

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <h:hermes xmlns:h="http://www.cl.cam.ac.uk/opera/hermes">
3   <advertisement>
4     <advertise typename="LocationEvent"/>
5   </advertisement>
6 </h:hermes>
```

Figure 4.11: An XML definition of an advertisement message

```
<?xml version="1.0" encoding="UTF-8"?>
<h:hermes xmlns:h="http://www.cl.cam.ac.uk/opera/hermes">
  <subscription>
    <subscribe typename="LocationEvent">
      <typeattr>
        <xpath>child::*[child::id>3141 and child::position="FE02"]</xpath>
      </typeattr>
    </subscribe>
  </subscription>
</h:hermes>
```

Figure 4.12: An XML definition of a subscription message

Returns	**API Call**	**Parameters**
`XMLDocument`	*toXML*	`()`
`void`	*fromXML*	`(XMLDocument doc)`
`XMLDocument`	*toXMLSchema*	`()`

Table 4.9: The XML-to-event binding API

Publication Message. As shown in Figure 4.13, an event publication is an XML Schema instance document (lines 5–10) that is encapsulated in a publication message. The event type to be published is given in line 4. Values for the event attributes in the `LocationEvent` type are assigned in lines 6–9 using the event attribute elements from the XML Schema definition for this event type.

Programming Language Integration

The communication between event clients and their hosting event brokers in Hermes uses XML-defined messages similar to the inter-broker messages presented above. Rather than exposing XML directly to the application, the Java implementation of event clients includes a binding layer between Java objects and XML messages. The *XML-to-event binding layer* supports the transparent translation of event publications to Java objects and vice versa so that the application programmer does not have to work with the XML representation of events.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <h:hermes xmlns:h="http://www.cl.cam.ac.uk/opera/hermes">
 3    <publication routing="typeAttr">
 4      <publish typename="LocationEvent">
 5        <t:LocationEvent xmlns:t="http://www.cl.cam.ac.uk/opera/hermes/type">
 6          <source>ActiveBat314</source>
 7          <id>1234</id>
 8          <timestamp>2003-06-20T12:00:00.000-00:00</timestamp>
 9          <position>FE02</position>
10        </t:LocationEvent>
11      </publish>
12    </publication>
13  </h:hermes>
```

Figure 4.13: An XML definition of a publication message

A Java class that represents an event publication in Hermes extends the `XMLDynamicBinder` abstract class, which provides the XML binding functionality for events. Its API, given in Table 4.9, has methods that are implemented using the Java support for structural reflection. A call to the `toXML` method returns an XML Schema instance document that is an event publication message with values assigned to the event attributes, as it was shown in Figure 4.13. When a client receives a callback with an event publication, the method `fromXML` reverses the process and assigns the data from the publication to a Java object. The last method `toXMLSchema` helps create new event types by translating a Java class definition to an XML Schema document that defines a new event type. This is illustrated in Figure 4.14, showing a Java class declaration and the corresponding XML Schema for an event type. Note that a `fromXMLSchema` method cannot be provided in Java because the language does not support the construction of new classes at runtime.

4.4.3 Middleware Service Extensions

The architecture of Hermes supports a services layer that provides extensions to event brokers. Our prototype implementation is flexible enough to support the three higher-level services for congestion control, composite event detection, and security that will be described in the remaining chapters of this thesis.

The Hermes implementation follows a modular design with clearly defined internal interfaces between modules. Plug-in modules that act as interceptors can be added to interfaces in order to change the calls between modules in an event broker. For services, such as congestion control and security, that depend on low-level access to messages, this plug-in approach is used to place service modules into the datapath of messages in the event broker. For example, it is possible to intercept calls of the Pan callback API before they reach the event routing layers for encryption. The XML-based approach for messaging in Hermes facilitates the addition of new message types to Hermes, such as the two new inter-broker messages to detect and react to congestion for the congestion control service in Chapter 6. Similarly, an extension to Hermes for event federations [Hom02] uses new message types to set up contracts between event brokers in different event domains.

4.5 Summary

In this chapter we described Hermes, our event-based middleware for large-scale distributed systems. We started with an overview of the layered overlay networks in Hermes. This layered

```
<xsd:complexType name="BaseEventType">
  <xsd:sequence>
    <xsd:element name="id"
      type="xsd:long" minOccurs="0"/>
    <xsd:element name="publisher"
      type="xsd:string" minOccurs="0"/>
    <xsd:element name="timestamp"
      type="xsd:dateTime" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

```
public class BaseEventType {

  public long id;

  public String publisher;

  public Date timestamp;


}
```

Figure 4.14: The mapping between an XML event type and a Java class

approach manifested itself in the description of the Hermes architecture that contains six layers. The architecture is realised in the Hermes event broker, whose inter-broker API was presented. We also introduced rendezvous nodes, which are special event brokers that help construct event dissemination trees. The second Hermes component is event clients, and the client APIs for event publishers and event subscribers were explained.

Hermes supports two different event routing algorithms, type-based routing that provides subscriptions depending on the event type, and type- and attribute-based routing that also includes content-based filtering expressions on the event attributes. The two algorithms were described by first listing the four types of messages that are exchanged between event brokers and then presenting the data structures maintained at event brokers. After that, the two routing algorithms were explained in more detail, with extensions to support event type inheritance and fault-tolerant routing. A description of how to implement both routing algorithms in a single system was given in pseudo code. The rest of the chapter focused on our prototype implementation of Hermes, in particular the implementation of the peer-to-peer routing layer, called Pan, and the XML message formats used by the event brokers. The next chapter will verify our claims with respect to scalability of Hermes's routing algorithm by presenting simulation results.

5 Evaluation

During the presentation of the Hermes event routing algorithms in the previous chapter, we made claims regarding their scalability and efficiency. In this chapter we provide a performance evaluation of Hermes routing to substantiate our claims with evidence from simulations [PB03a]. The evaluation is done through a number of experiments in a distributed systems simulator that investigate routing efficiency and routing state in the system. The experiments show that Hermes is scalable and efficient when compared against a leading content-based routing algorithm for event dissemination which does not use peer-to-peer routing techniques. In particular, peer-to-peer overlay routing in Hermes does not reduce the efficiency of routing, but rather improves certain metrics, such as the latency of event delivery, because of an improved mapping of the overlay broker network onto the physical network.

This chapter is organised as follows. In the next section we give an overview of our strategy for evaluating Hermes. The simulation environment used for the experiments is described in Section 5.2, with details on Internet topology generation, our distributed systems simulator DSSim, and the implementation of CovAdv, a standard content-based routing algorithm. Section 5.3 focuses on the setup of the experiments and lists the main simulational parameters. The actual four experiments are introduced and motivated in Section 5.4, including a presentation of the results obtained.

5.1 Overview

The implementation of an event-based middleware, such as Hermes, can be evaluated with performance measurements using a real, deployed application or with experiments in a simulator. Although an actual deployment results in a more realistic evaluation, it is more difficult to instrument since a large-scale distributed system has substantial resource requirements in terms of node count and network bandwidth. Instead, we decided to set up experiments in a distributed systems simulator that supports simulations with thousands of Hermes event brokers and millions of messages. When simulating any large-scale system, it is important to ground the simulation on realistic assumptions, such as a good Internet topology model [BP96, FP01].

Moreover, the usage pattern for the event-based middleware should be derived from realistic application scenarios.

The goal of this evaluation is to point out the advantages of the dynamic overlay broker network for event dissemination maintained by the peer-to-peer routing layer in Hermes, compared to an approach with a static overlay broker network, as advocated by Siena. The quality of the mapping of the logical overlay broker network onto the physical network topology is an important factor for the overall efficiency of event routing in the system. A publish/subscribe system that constructs event dissemination trees in the logical overlay network of event brokers without taking account of the underlying physical network topology will have poor performance and little fault-tolerance when disseminating events. The content-based routing algorithm should build an optimal event dissemination tree for each publisher-hosting broker in the overlay routing network that encompasses the subscriber-hosting brokers of all interested event subscribers. We have chosen three *cost metrics* that evaluate the efficiency of event dissemination trees, and thus event routing in an event-based middleware.

Latency. Many applications require event dissemination trees that minimise the time until all interested subscribers have received a given publication. Therefore, routing paths in an event dissemination tree should attempt to reduce the latency of message routing.

Hop Count. Since content-based routing is done by application-level event brokers, an increased hop count can be expensive in terms of the latency experienced due to event processing at event brokers, the required processing power at event brokers, and the increased probability of event broker failure. Minimising the number of event brokers in an event dissemination tree should therefore be desirable.

Bandwidth. An advantage of an overlay broker network is that it can share a physical network with other applications. This means that event dissemination trees should use as little bandwidth as possible leaving more for other applications. Bandwidth should be saved by reducing the number of messages sent in the system.

To evaluate the efficiency of the Hermes routing algorithm, we compare it against our implementation of Siena's content-based routing algorithm, called CovAdv. Such a relative performance comparison is more meaningful than measured absolute values. We have chosen Siena for the comparison because its routing algorithm is similar to algorithms used by many other systems and it can therefore be regarded as a typical publish/subscribe system. Since Siena supports content-based filtering of events, the experiments compare it to type- and attribute-based routing in Hermes with respect to event dissemination latency, physical network utilisation, and routing state maintained at event brokers. The experiments in this chapter do not involve failure or Hermes's fault-tolerance mechanisms because we are interested here in the routing behaviour of both algorithms under optimal conditions.

A further advantage of simulation is that it helps ensure the correctness of the routing algorithms and their implementation in an event-based middleware. During the development of Hermes, the simulator was an invaluable tool to simulate small Hermes deployments and verify that the routing data structures maintained at event brokers behaved as anticipated.

5.2 Simulation Environment

The simulation environment should be powerful enough to support measurements of latency, hop count, and bandwidth in the experiments. As a result, the simulator must be aware of the

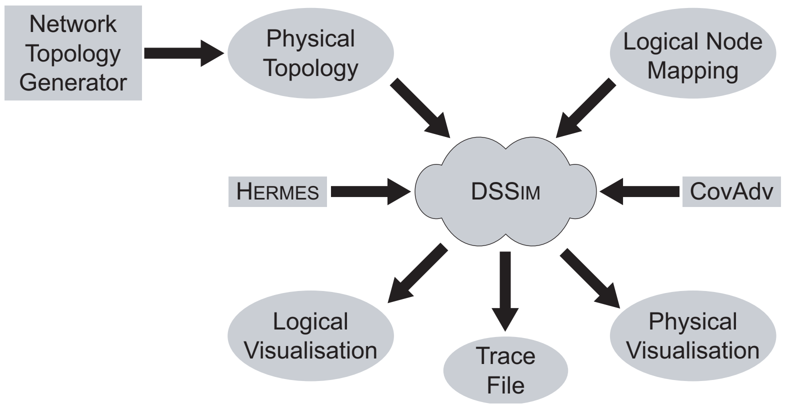


Figure 5.1: The DSSIM simulation environment

underlying physical network topology so that multi-hop message routing is associated with the latency of a path through network-level routers. Previous efforts to evaluate publish/subscribe systems through simulation [Car98, MFGB02] often only considered the overlay network level and hence could not determine the quality of the overlay network mapping.

One approach for a realistic simulation would be to use a standard network simulator, such as ns-2 [MF99]. However, we found that the scalability of network simulators is limited to a moderate number of nodes because of the complexity of their realistic network model. Since our evaluation is more concerned with the efficiency of the distributed routing algorithms rather than the lower-level network protocols, we decided to implement our own distributed systems simulator DSSIM.

The simulation environment used for the rest of this chapter is depicted in Figure 5.1. It consists of a network topology generator that creates realistic large-scale Internet topologies for the experiments. The physical network topology is used as input to the distributed systems simulator DSSIM that executes the implementation of the middleware. A logical node mapping specifies the location of event brokers and clients in the physical network. The simulator is then capable of executing two different event routing algorithms, the HERMES type- and attribute-based routing algorithm described in Section 4.3.5, and the CovAdv routing algorithm described below, which is an implementation of SIENA routing with a coverage relation and advertisements. The simulator outputs a trace file with statistics gathered during the experiment run that can be plotted as graphs. In addition, the simulator can display a logical or physical visualisation of the currently running simulation on the screen.

5.2.1 Internet Topology Generation

Several topology models are supported by DSSIM for the generation of physical networks. Simple topologies, such as a euclidean plane and a spherical topology, are mainly used for testing the correctness of the simulator and of the event routing algorithms. For our experiments, we adopted the *transit-stub model* [ZCB96] to generate large-scale topologies that resemble the structure of the Internet. A transit-stub topology consists of multiple domains that represent Internet autonomous systems. A domain is a *stub domain* if it never caries any transit traffic, otherwise it is called a *transit domain*. The purpose of transit domains is to interconnect stub domains efficiently, thus forming an Internet backbone for other autonomous systems.

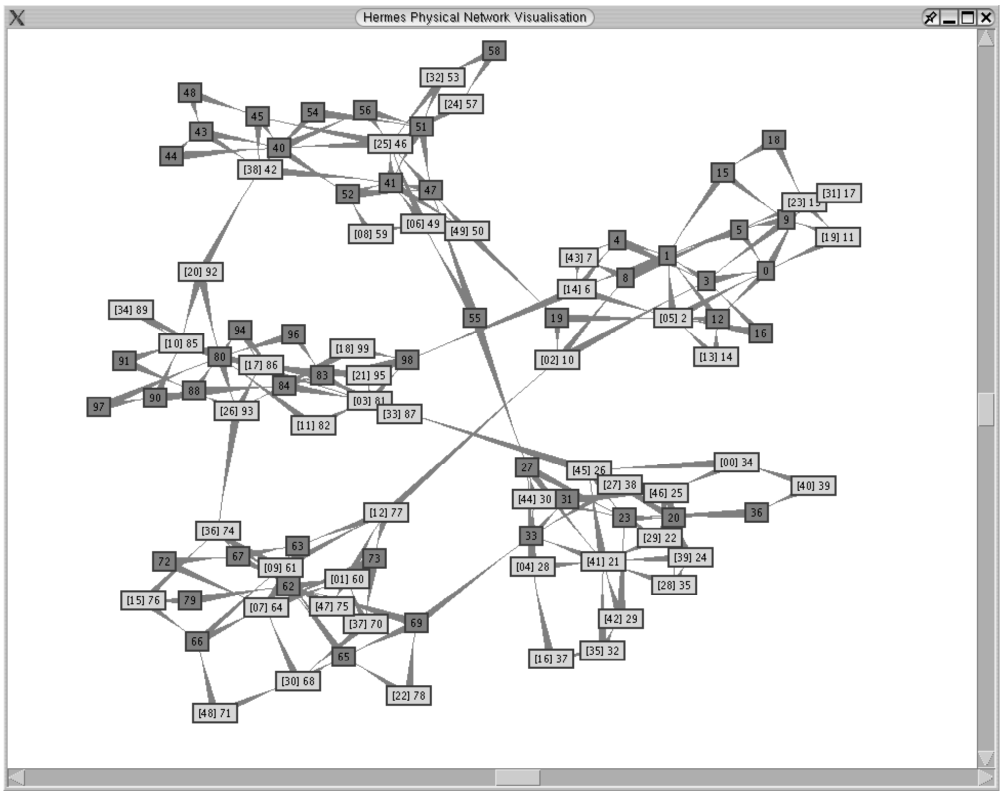


Figure 5.2: A transit-stub topology with five autonomous systems

We use BRITE [MLMB01] as a transit-stub topology generator. It allows the customised creation of transit-stub topologies and assigns latency and bandwidth values to links so that inter-domain (WAN) links usually have higher latency and lower bandwidth. A transit-stub topology with 5 autonomous systems each having 20 nodes is shown in Figure 5.2. The latency of physical links varies from 10 ms to 1000 ms. A logical node mapping can now be used to place event brokers on the physical nodes in the topology.

5.2.2 DSSim

The *distributed systems simulator* DSSim is implemented in Java as a single-threaded discrete event simulator [BCM87]. The primary design consideration for DSSim was scalability. Therefore, it supports simulations with 10^5 physical nodes while only having moderate CPU and memory requirements. DSSim models the latency and the hop count of routing messages in the physical network, and it also maintains a measure for bandwidth consumption. Note that congestion caused by bandwidth-limited links is not modelled in DSSim. For now, we assume that event publication messages are sufficiently small to never saturate a physical link leading to congestion. This assumption will be revisited in Chapter 6 when we address congestion in an event-based middleware and introduce a congestion control mechanism.

DSSim distinguishes between a *physical network topology* generated by the topology generator and a *logical network topology* obtained after mapping application-level nodes that execute a distributed algorithm onto the physical nodes. Logical nodes communicate through message passing, and DSSim simulates the routing of messages in the physical network topology. Expen-

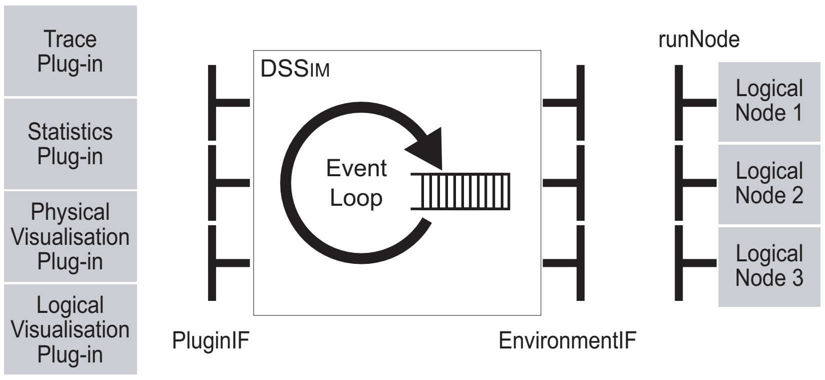
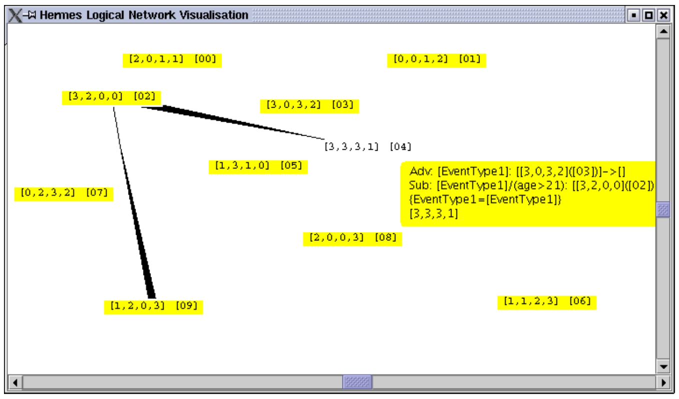


Figure 5.3: The architecture of DSSIM

sive routing table management frequently limits the scalability of network simulators. DSSIM uses a hierarchical two-level distance vector routing algorithm that mimics Internet routing between and within autonomous systems.

An overview of the internal architecture of DSSIM is provided in Figure 5.3. The core of DSSIM is an event loop that takes simulation events from a time-ordered event queue and executes them. A simulation event is, for example, the request of a logical node to send a message, or the routing of a message by a physical node. Events are added to the event queue by logical nodes through the `EnvironmentIF` interface. This interface enables a logical node to interact with its environment and handle messaging, monitoring, and time aspects during the simulation. The execution of logical nodes is scheduled by the simulator, which calls a `runNode` method whenever a particular node is supposed to run.

The functionality of the simulator can be extended with plug-ins that are inserted into the event loop and can process simulation events. Currently there are four plug-ins implemented in DSSIM: a *trace plug-in* that records all simulation events in a file for later replay, a *statistics plug-in* that gathers statistics during the simulation, and two *visualisation plug-ins*. The visualisation plug-ins allow the display of a graphical representation of the logical and physical network topologies and can visualise the routing of messages and the internal state of logical nodes. In Figure 5.4 an example visualisation is given showing the routing state at a HERMES event broker during an experiment run. The visualisation plug-ins proved to be especially useful for the debugging of HERMES.



Adv: [EventType1]: [[3,0,3,2]([03])]->[]
Sub: [EventType1]/(age>21): [[3,2,0,0]([02])
{EventType1=[EventType1]}
[3,3,3,1]

Figure 5.4: Logical visualisation in DSSIM

5.2.3 CovAdv

For comparison, we implemented the Siena routing algorithm described in Section 2.2.2 in DSSim. Our implementation is called *CovAdv* and exports the same API for event clients as Hermes. It uses advertisement messages sent by publisher-hosting brokers, subscription messages sent by subscriber-hosting brokers that follow the reverse path of advertisements, and publication messages that follow the reverse path of subscriptions. Covering between advertisements and subscriptions in routing tables is implemented using shared code between Hermes and CovAdv. As CovAdv does not support rendezvous nodes on top of a peer-to-peer routing substrate, advertisement messages are broadcast to all event brokers unless they are covered by previous advertisements.

The overlay broker network in CovAdv follows Siena's approach of an *acyclic peer-to-peer topology* [Car98] that interconnects event brokers in an acyclic undirected graph. We also considered testing the *generic peer-to-peer topology* in Siena that permits cycles in the overlay broker network, but this would have required new extensions to the Siena routing algorithm in order to patch up routing tables after topology changes. These topology changes occur during normal operation when more efficient routing paths are discovered by the incremental distance-vector routing algorithm used by Siena. Instead, the logical overlay network of CovAdv event brokers is computed to be a minimum spanning tree with respect to network latency. A disadvantage of this approach is that there is no redundancy in the neighbouring broker relation in CovAdv.

At deployment time the sets of neighbouring event brokers in CovAdv have to be statically defined by a system administrator. An administrator at one site may not be aware of all event brokers in the event-based middleware deployment, making it difficult to ensure global properties, such as acyclicity or being a minimum spanning tree. In addition, the calculation of global overlay network properties can be computationally expensive. Since the CovAdv overlay network is static, it cannot adapt to failure and system evolution. Because of the requirement of administrability from Section 3.2.3, Hermes does not have these restrictions.

5.3 Experimental Setup

All experiments that will be described in the next section were carried out in DSSim using a transit-stub network topology with $N_T = 1000$ physical nodes partitioned into $N_{\text{AS}} = 10$ autonomous systems with $N_{\text{as}} = 100$ nodes each. Although this physical network size is well below the maximum supported by DSSim, we verified that the obtained results can be scaled up and are representative of larger networks. Event brokers, publishers, and subscribers were assigned randomly to physical nodes unless stated otherwise. The Pan peer-to-peer routing layer used nodeIDs with base $b_{\mathsf{Pan}} = 4$ and a leaf set size of $l_{\mathsf{Pan}} = 4$. Each experiment was repeated $i = 5$ times and arithmetic means were taken of any measured values. Table 5.1 lists all simulation parameters for the experiments.

In the experiments, we varied the number of event brokers n_E, event publishers n_P, event subscribers n_S, and event types n_τ in the system. Every event publisher published n_{P_τ} different event types that were randomly selected from the total number of n_τ event types. Only event types that were going to be published were also advertised. Correspondingly, every event subscriber subscribed to n_{P_τ} different event types either through a Hermes type- and attribute-based subscription or a CovAdv subscription, depending on the routing algorithm in the experiment run. For simplicity, filtering was only performed on event types and not on event

Parameter	Description	Value
i	number of experiment runs	5
N_T	number of physical nodes in network	1000
N_{AS}	number of autonomous systems in network	10
N_{as}	number of physical nodes per autonomous system	100
b_{Pan}	base for PAN nodeIDs	4
l_{Pan}	leaf set size in PAN	4
n_E	number of event brokers	$50 \ldots 1000$
n_P	number of event publishers	$10 \ldots 5000$
n_S	number of event subscribers	$10 \ldots 5000$
$n_{\mathcal{T}}$	number of event types	$1, 10$
$n_{P\mathcal{T}}$	number of event types per publisher	$5, 10$
$n_{S\mathcal{T}}$	number of event types per subscriber	$5, 10$

Table 5.1: Simulation parameters for the experiments

attributes but this had no effect on the performed experiments except for the introduction of more rendezvous nodes in HERMES.

Although the number of event brokers remains at a moderate scale, this is more realistic of current middleware deployments because such an overlay broker network can support a large number of event clients. Since our evaluation focuses on overlay routing, we also limited the number of event clients in the experiments because event routing is rather influenced by client-hosting brokers and not event clients.

The HERMES and CovAdv experiments differed in the way the overlay broker network was constructed. In the case of HERMES, event brokers were sequentially added to the simulated system. The bootstrapping broker chosen for each new event broker was the closest (in terms of network latency) already existing event broker in the physical topology. This is a realistic deployment strategy, since a system administrator only needs to know of a single, close-by HERMES event broker when adding a new broker to the system.

For CovAdv, we precomputed a minimum spanning tree of event brokers and used this for the neighbouring broker sets. Although this gave CovAdv routing the advantage of an optimal overlay broker network in terms of latency, it only directly affected the experiments that were concerned with the latency of event delivery. In all other experiments, it ensured that CovAdv routing would not suffer too much under a poor quality overlay broker network, thus allowing a better comparison with HERMES routing.

5.4 Experiments

In this section we describe the four experiments E1–E4 that compare the performance of HERMES routing with CovAdv routing and investigate different aspects of event dissemination in an event-based middleware. The experiments were chosen to evaluate event routing in terms of the cost metrics stated in Section 5.1. We will present the measured statistics in a series of plots and then discuss the obtained results, referring back to the operation of the HERMES and CovAdv routing algorithms.

The first experiment E1 looks at the efficiency of event routing in terms of latency and number of hops in the physical network. This measures the quality of the overlay broker network constructed by Hermes for event dissemination. The second experiment E2 shows the amount of state kept in routing tables at event brokers for both algorithms. The distribution of routing table state across event brokers is examined in experiment E3. Finally, experiment E4 considers the bandwidth usage in the event-based middleware by measuring the total number of messages sent by both algorithms when publishing a fixed number of events.

5.4.1 E1: Routing Efficiency

The efficiency of routing in Hermes and CovAdv is evaluated in experiment E1 using two metrics to measure the efficiency of event dissemination. The *average latency per event notification* $\bar{l}$ is calculated as

$$\bar{l} = \frac{\Sigma l_i}{n_{\text{total}}} \tag{5.1}$$

where l_i is the latency for the delivery of a given event publication to all interested subscribers and n_{total} is the total number of delivered publications. Note that this metric is independent of the size of the event dissemination tree and states how efficiently the event dissemination tree utilises the underlying physical network with respect to latency. Many distributed applications benefit from a low delay between publication of an event and notification of a client. For the second metric, the *average hop count per event notification* $\bar{h}$ is defined as

$$\bar{h} = \frac{\Sigma h_i}{n_{\text{total}}} \tag{5.2}$$

where h_i is the hop count for the delivery of a given event publication to all interested subscribers. Physical nodes in the network topology that may host a logical node, such as an event broker, count towards the hop count. Therefore this metric is the average number of hops until an event is delivered and favours event dissemination trees that involve a smaller number of physical and thus logical nodes. A smaller average hop count has the advantage that less processing is performed at event brokers.

In experiment E1 the number of event brokers was $n_E = 500$ and the number of event publishers was $n_P = 10$. Only a single event type, $n_{\mathcal{T}} = 1$, existed in the system and all event clients operated on this type. The number of event subscribers varied from $n_S = 10 \dots 500$. At most one subscriber was hosted by an event broker because the shape of event dissemination trees is only affected by subscriber-hosting brokers and not by event subscribers with equivalent subscriptions connected to the same broker. The experiment was repeated three times with the two routing algorithms and different strategies for creating the overlay broker network.

Hermes with closest event broker. The closest, already existing event broker in the overlay network is chosen as the bootstrapping broker for a new event broker that is added to the system. Proximity is defined in terms of network latency.

CovAdv with closest event broker. The same strategy as above is used to build the overlay broker network in CovAdv.

CovAdv with minimum spanning tree. A minimum spanning tree in terms of latency between event brokers is precomputed for the neighbouring broker relation in CovAdv.

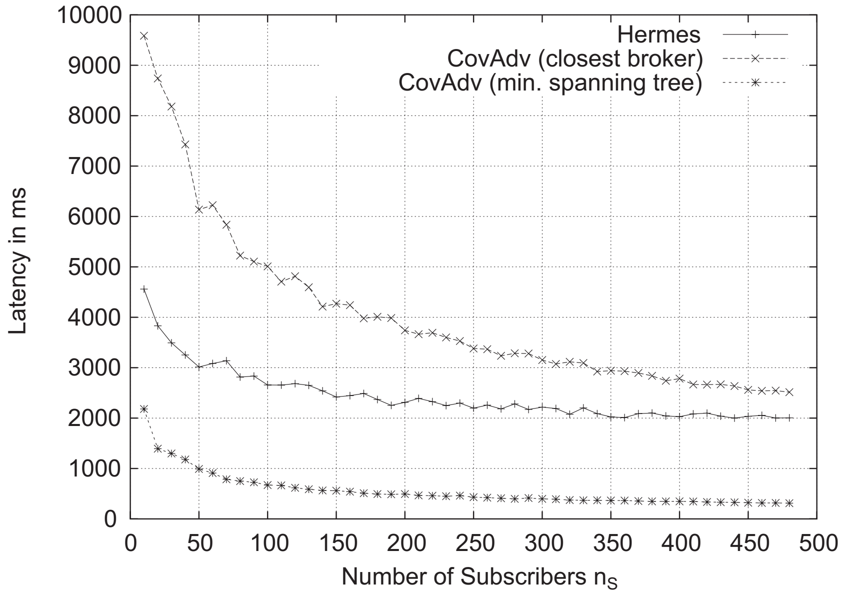


Figure 5.5: E1: Latency per event versus number of event subscribers

The plot in Figure 5.5 shows the average latency per event notification $\bar{l}$ in milliseconds as a function of the number of event subscribers. As expected, the latency per notification decreases as more subscribers are added because event dissemination trees cover a larger portion of event brokers. After a certain number of event subscribers, the system reaches a saturated state in all three cases, where all event brokers have to receive a published event. In a realistic deployment of CovAdv with the closest event broker for the neighbouring broker set, CovAdv exhibits poor latency behaviour. Especially when the system is sparsely populated with subscriber-hosting brokers, CovAdv shows the largest latency times since its overlay broker network does not adequately reflect the latency of the links in the physical network when creating event dissemination trees. In contrast, HERMES with the closest event broker results in a good compromise with respect to event latency and is less dependent on the number of subscriber-hosting brokers. Unsurprisingly the lowest latency is achieved by CovAdv with a precomputed minimum spanning tree, but this requires the highest computational and administrative overhead.

The average hop count per event notification $\bar{h}$ is plotted in Figure 5.6. The graphs for all three cases reach a saturated hop count of one when an event subscriber is notified after every routing hop. HERMES routing exhibits a very efficient behaviour with the smallest number of routing hops. This is mainly due to the larger size of the neighbouring broker sets that are automatically maintained by PAN. Note that even when the overlay broker network is sparsely populated with subscriber-hosting brokers ($n_S = 10$), HERMES routing achieves a low hop count of $\log_{16}(500) = 2.24$ on average. CovAdv with a minimum spanning tree has the highest average hop count because its minimum spanning tree is constructed from many low latency links.

5.4.2 E2: Space Efficiency

The second experiment E2 focuses on the state stored in the event-based middleware that is required for event dissemination for both routing algorithms. We measure the amount of state as the number of entries in advertisement and subscription routing tables at event brokers. Reducing maintained state at event brokers has the benefit that fewer resources are needed and

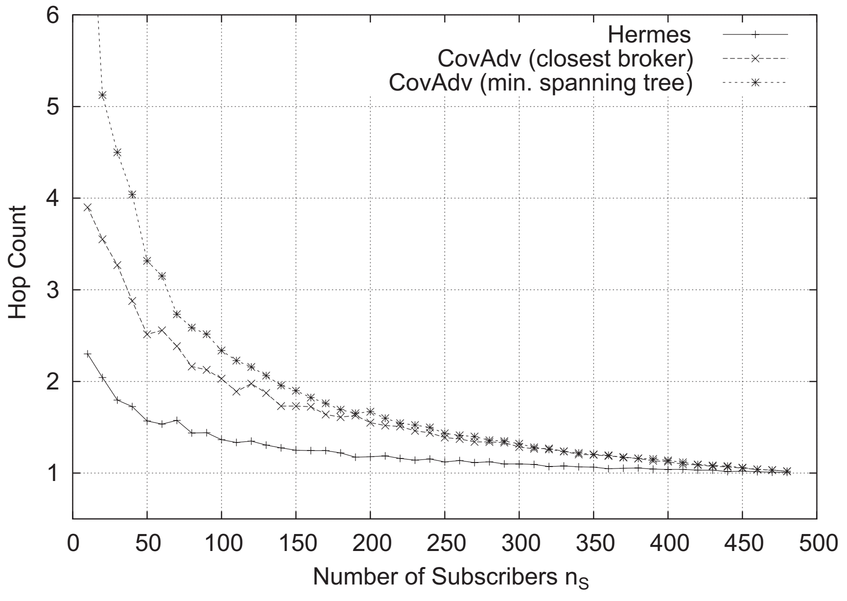


Figure 5.6: E1: Hop count per event versus number of event subscribers

recovery after failure is simplified. In addition, increase in state as a function of the number of event brokers or clients is one of the factors that determines the scalability of an event-based middleware.

To evaluate space efficiency, we varied either the number of event brokers, event publishers, or event subscribers while keeping the other parameters fixed and observed how this affected the number of entries in advertisement and subscription routing tables. There were $n_\mathcal{T} = 10$ event types in the system, and each event client used $n_{P_\mathcal{T}} = n_{S_\mathcal{T}} = 5$ event types.

The result of the first experiment run with $n_E = 500$ event brokers, $n_P = 10$ event publishers, and the number of event subscribers ranging from $n_S = 100 \ldots 5000$ is given in Figure 5.7. Considering entries in advertisement routing tables, Hermes creates a fraction of the state needed by CovAdv because event advertisements are only propagated to rendezvous nodes instead of all event brokers in the system. The size of advertisement routing tables stays constant because the number of event publishers remains unchanged. Due to Hermes' more efficient routing in the overlay broker network, it can better exploit subscription coverage and therefore uses slightly less state in subscription routing tables than CovAdv. As more subscribers are added, both algorithms converge to the same number of entries because the routing tables are completely filled when every subscriber-hosting broker has event subscribers subscribing to all events.

In the second experiment run in Figure 5.8, the number of event publishers varied from $n_P = 100 \ldots 5000$, whereas the number of event brokers ($n_E = 500$) and event subscribers ($n_S = 100$) were kept constant. The count of entries in advertisement routing tables in CovAdv is unchanged because broadcasts of advertisement messages immediately create a complete set of entries at all event brokers. In contrast, entries for advertisements in Hermes scale sub-linearly with the number of event publishers in the system. Since Hermes uses type- and attribute-based subscriptions, which create more filtering state as more event publishers are added, the amount of state in subscription routing tables increases sub-linearly, as well. Subscription state in CovAdv scales less well because of its less efficient overlay routing.

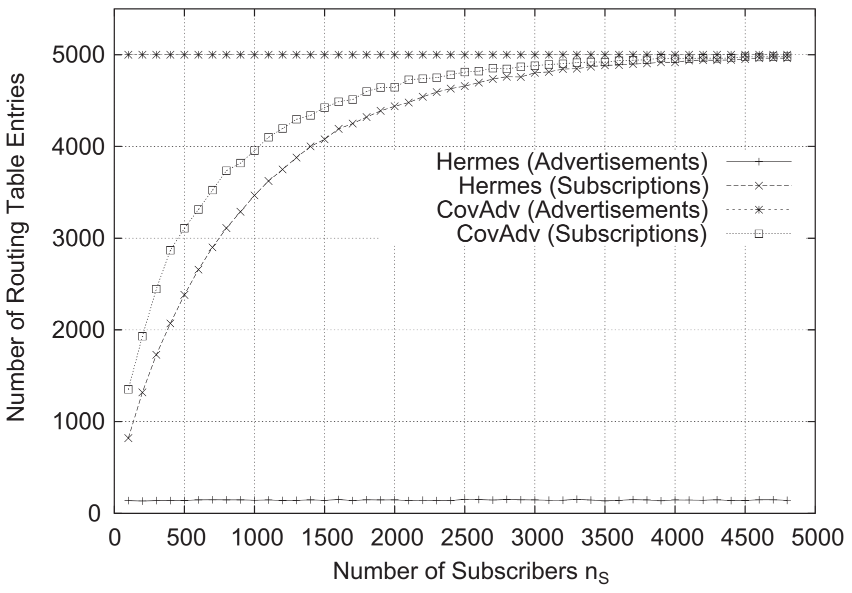


Figure 5.7: E2: Routing tables entries versus number of event subscribers

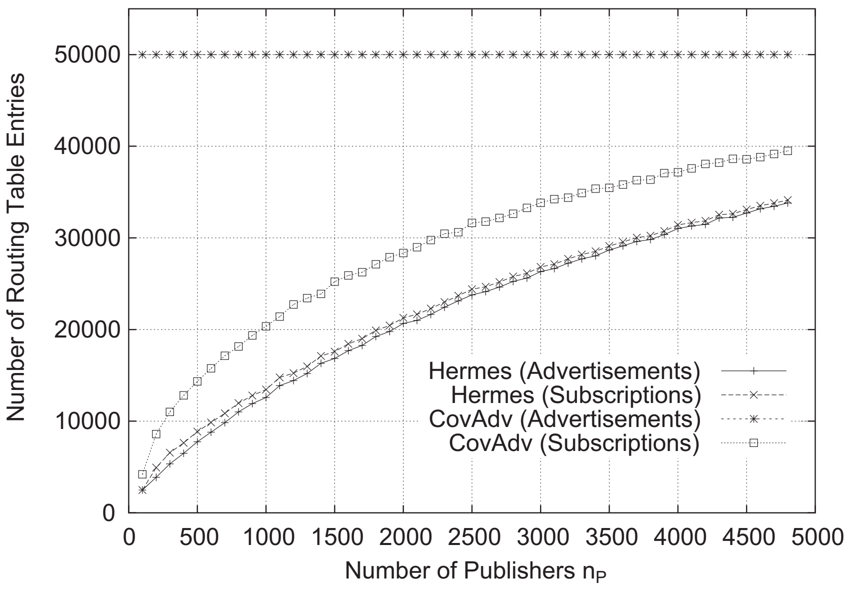


Figure 5.8: E2: Routing tables entries versus number of event publishers

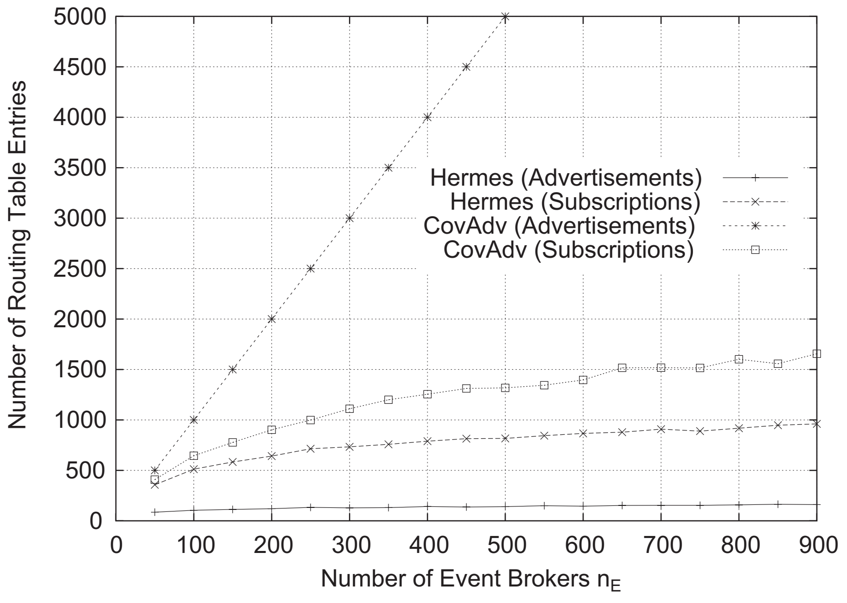


Figure 5.9: E2: Routing table entries versus number of event brokers

The scalability of the overlay broker network with respect to the number of event brokers was investigated in the third experiment run. For this, the number of event brokers was altered from $n_E = 50 \dots 1000$ with a fixed number of event publishers ($n_P = 10$) and event subscribers ($n_S = 100$). As can be seen in the plot in Figure 5.9, most state in routing tables in CovAdv is created by advertisements that scale linearly with the number of event brokers. Since advertisements only reach rendezvous nodes in HERMES, far less state is required. The increase in subscription state is similar for both systems although the overlay broker network in HERMES scales slightly more favourably.

5.4.3 E3: Space Distribution

In an event-based middleware not only the amount of state in the overlay broker network is important, but also the distribution of state across event brokers. A system that distributes state uniformly across all event brokers involves a larger number of components and is therefore more prone to failure. The principle of local routing suggests that only event brokers on the routing path of an event dissemination tree should be required to maintain state. Therefore, experiment E3 surveys the distribution of routing tables entries in HERMES and CovAdv.

The experiment was run with fixed values for all parameters. The number of event brokers was $n_E = 500$ with $n_P = 10$ event publishers and $n_S = 1000$ event subscribers. There were $n_\tau = 10$ event types and each client used $n_{P_\tau} = n_{S_\tau} = 5$. The total number of routing table entries at an event broker is the sum of entries in advertisement and subscription routing tables.

The distribution of routing table entries is presented in Figure 5.10. Routing state in CovAdv is spread out across a large fraction of event brokers. Almost the majority of event brokers has a maximum of 20 entries in routing tables and no single broker has less than 10 entries. As a result, a random event broker failure is likely to result in the loss of significant routing state. In contrast, HERMES clearly has a better distribution of routing table state. The most common number of routing table entries is 10 with 100 event brokers. No event broker has more than 15 entries and 40 event brokers with empty routing tables are not involved in routing at all because

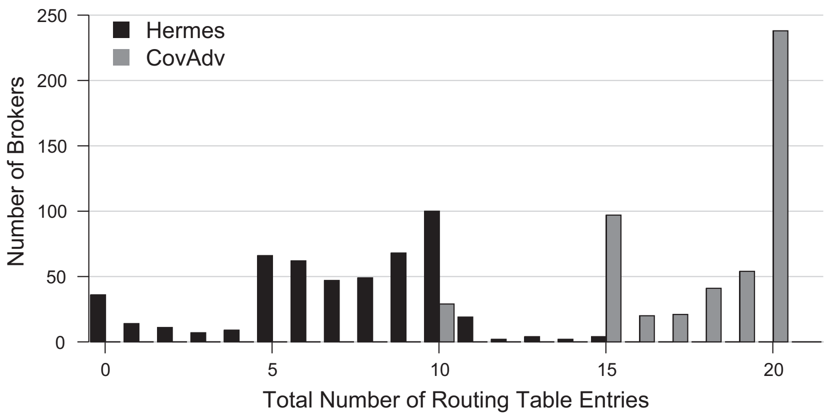


Figure 5.10: E3: Distribution of routing tables entries at event brokers

they are not part of any event dissemination trees. This result is mainly due to the fact that routing in Hermes does not broadcast messages in the system.

5.4.4 E4: Message Complexity

The final experiment E4 examines a bandwidth cost metric in an event-based middleware. Instead of measuring the actual bandwidth used by the routing algorithms during event dissemination, network utilisation is expressed in terms of message counts of advertisement, subscription, and publication messages. Fewer exchanged messages means that more network resources are available for other applications and less processing has to be done at event brokers.

The same number of event brokers ($n_E = 500$) was used as in the previous experiment. However, more event types were available in the system ($n_\mathcal{T} = 100$) in order to reduce coverage between messages and hence exchange more messages. Clients also used a wider range of distinct event types ($n_{P_\mathcal{T}} = n_{S_\mathcal{T}} = 10$). We performed two runs of the experiment, varying either the number of event publishers n_P or event subscribers n_S and observed the counts for each message type.

In Figure 5.11 the plot of messages sent as a function of the number of event subscribers in the range from $n_S = 100 \ldots 5000$ is shown. There were $n_P = 100$ event publishers in the simulation. The Hermes routing algorithm sends fewer publication messages than CovAdv due to its more efficient event dissemination trees. However, an additional cost in terms of subscription messages comes from subscriptions having to reach rendezvous nodes in Hermes. As a consequence, Hermes sends more subscriptions messages when there are over approx. 1200 subscribers. As discussed before, the counts of advertisement messages remain constant with substantially more messages being sent by CovAdv.

In the second experiment run given in Figure 5.12, there were $n_S = 10$ event subscribers and the number of event publishers was changing ($n_P = 100 \ldots 5000$). As with a variable number of event subscribers, fewer publication messages are sent by Hermes. Again, there is a threshold of approx. 5000 publishers after which Hermes sends more advertisement messages than CovAdv because of rendezvous node routing.

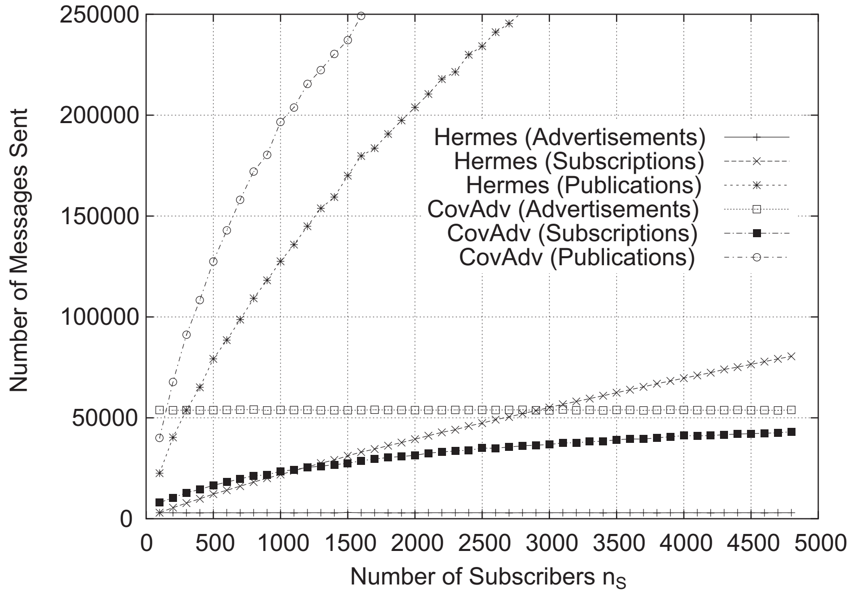


Figure 5.11: E4: Number of messages versus number of event subscribers

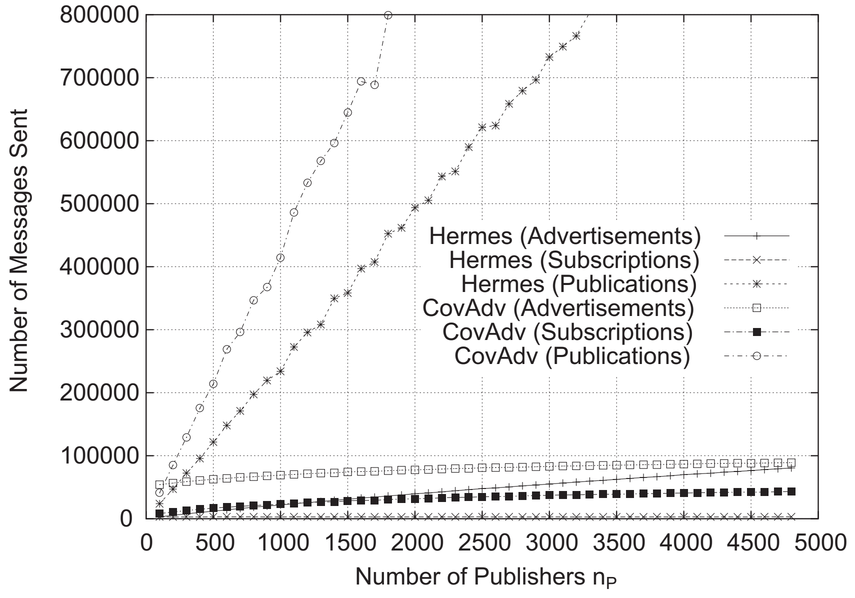


Figure 5.12: E4: Number of messages versus number of event publishers

5.5 Summary

This chapter has presented the simulational evaluation of event routing in Hermes. It started with a motivation of our evaluation methodology by making a case for a simulational approach that compared the routing algorithm of Hermes to another, well-known content-based routing algorithm. Next cost metrics for the experiments in terms of latency, hop count, and bandwidth were introduced. We then described the simulation environment based on DSSim, a large-scale distributed systems simulator with a transit-stub topology generator and an implementation of the Siena routing algorithm called CovAdv. The actual evaluation was carried out in the form of four experiments that investigated different aspects of event dissemination.

The first experiment showed the quality of the constructed overlay broker network in Hermes used to build event dissemination trees. Both the average latency and hop count remain low when compared to CovAdv. In general, Hermes maintains less state in routing tables at brokers, which was demonstrated by the second experiment. The substantial saving in size of advertisement routing tables is due to rendezvous node routing in Hermes and subscription routing tables also benefit from the efficient overlay broker network. The results from the third experiment on state distribution in routing tables illustrated Hermes' approach of making routing decisions locally and only involving a small number of brokers. Finally, network utilisation in Hermes is lower because fewer messages are sent by brokers. This saving is largest for publication messages, which are the most common message type in an event-based middleware.

In summary, the Hermes peer-to-peer routing approach over a distributed hash table compares favourably with a traditional content-based routing algorithm, such as Siena routing. A Hermes event-based middleware does efficient event dissemination in terms of event routing decisions and resource usage and scales to a large number of event brokers and clients.

6 Congestion Control

The evaluation of Hermes event routing in the previous chapter was based on the assumption that event publication messages are negligible in size and therefore cannot saturate the available network bandwidth or processing power. However, this is not true in practice and an event-based middleware can suffer from congestion leading to a degradation of service to clients. In this chapter we propose a scalable congestion control mechanism [PB03b] that prevents the occurrence of congestion in a reliable publish/subscribe system. The mechanism is provided as a higher-level service for an event-based middleware according to the service model from Section 3.3.5. It consists of two algorithms, PDCC and SDCC, that are used in combination to address different aspects of congestion control in the event-based middleware.

To motivate the need for congestion control in an event-based middleware, we begin with an overview of the congestion control problem in the next section. From this, requirements for a mechanism to handle congestion are developed in Section 6.2. The main part of this chapter is the description of the two congestion control algorithms in Section 6.3. In Section 6.4 a prototype implementation is described, which is then used to experimentally evaluate the congestion control algorithms in Section 6.5. The chapter finishes with a discussion of related work in the area of congestion control.

6.1 The Congestion Control Problem

We argue that it is necessary to provide congestion control for overlay networks, such as the one established by an event-based middleware. In this section we illustrate the congestion control problem by giving examples of undesired behaviour due to congestion. *Congestion* occurs when there are not enough resources to sustain the rate at which event publishers send publication messages in an event-based middleware. We distinguish between two kinds of congestion,

1. *network congestion*, where the network bandwidth between event brokers is the limiting resource, and

2. *event broker congestion*, when the processing of messages at an event broker cannot cope with the data rate.

Both kinds of congestion may lead to the loss of messages at event brokers. Message loss is especially undesirable under guaranteed delivery semantics, as described by the reliability model in Section 3.3.4, because the resulting retransmission of messages worsens the level of congestion in the system. An event-based middleware suffers from *congestion collapse* when the message loss dominates its operation and prevents event clients from receiving any useful service.

Usually there are two reasons for congestion in an event-based middleware. In many cases, congestion is caused by the underprovisioning of the deployed middleware in terms of network bandwidth or processing power of event brokers so that the middleware cannot handle resource requirements of event dissemination during normal or peak operation. A second, more subtle cause for congestion is the temporary need for more resources as a result of recovery after a failure under guaranteed delivery semantics. We will give examples of both causes of congestion in the form of two experiments in Gryphon using its guaranteed delivery service, but the results are relevant to other publish/subscribe systems as well. Gryphon is an industrial-strength publish/subscribe system that was described in Section 2.2.2.

The experiments use the Gryphon overlay broker network depicted in Figure 6.1, consisting of a publisher-hosting broker PHB, an intermediate broker IB, and two subscriber-hosting brokers, SHB_1 and SHB_2. The PHB has four publishing endpoints (pubends), which represent ordered event streams from multiple event publishers hosted at the broker. As explained in Section 2.2.2, delivery guarantees in Gryphon are made with respect to the aggregated event publications in the event stream maintained in persistent storage at a pubend.

In the first example, the PHB publishes event publications at a rate of 500 msgs/s. The plot in Figure 6.2 shows the message rates observed at the PHB and both SHBs as a function of time. During the first 120 s, event dissemination is in a steady-state with 500 msgs/s being received at both SHBs. After that, a bandwidth restriction is imposed on the IB–SHB_1 link limiting it to about one third of the required bandwidth. Since the PHB continues sending event publications at an unchanged rate, outgoing buffer queues at the IB start to build up as can be seen in Figure 6.3, and eventually overflow causing message loss because the IB cannot send messages fast enough through the bandwidth-limited link. These lost publication messages need to be recovered and are therefore retransmitted, which further increases congestion resulting in congestion collapse with a drop of the message rate at SHB_1 to zero.

The second example, shown in Figure 6.4, demonstrates congestion caused by increased resource demand during recovery. Initially the bandwidth of the PHB–IB link is restricted to a value slightly higher than what is needed for steady-state operation. The IB–SHB_2 link is then brought down for 120 s. After the link has come back up, the system attempts to recover the lost

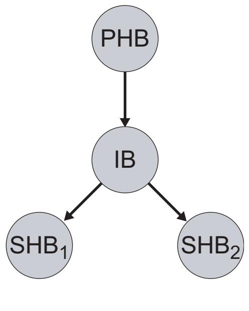


Figure 6.1: An overlay broker network topology with four event brokers

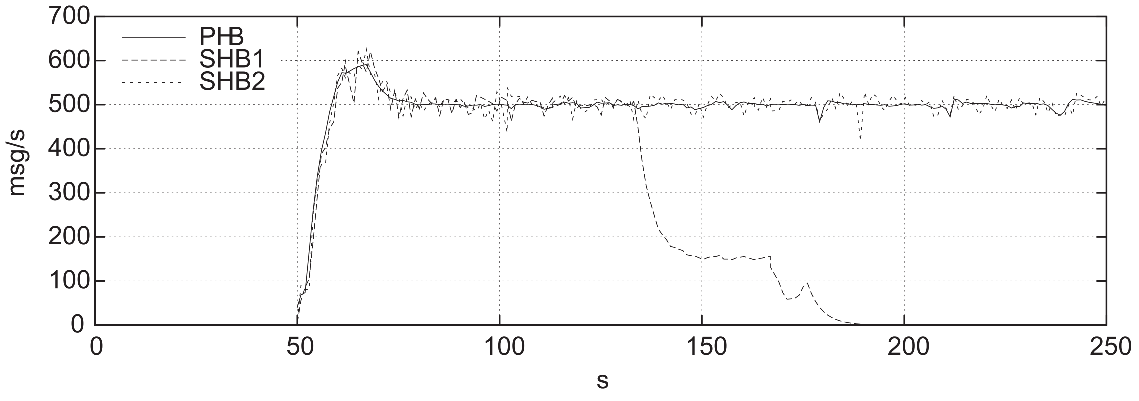


Figure 6.2: Congestion collapse with the IB–SHB$_1$ link restricted

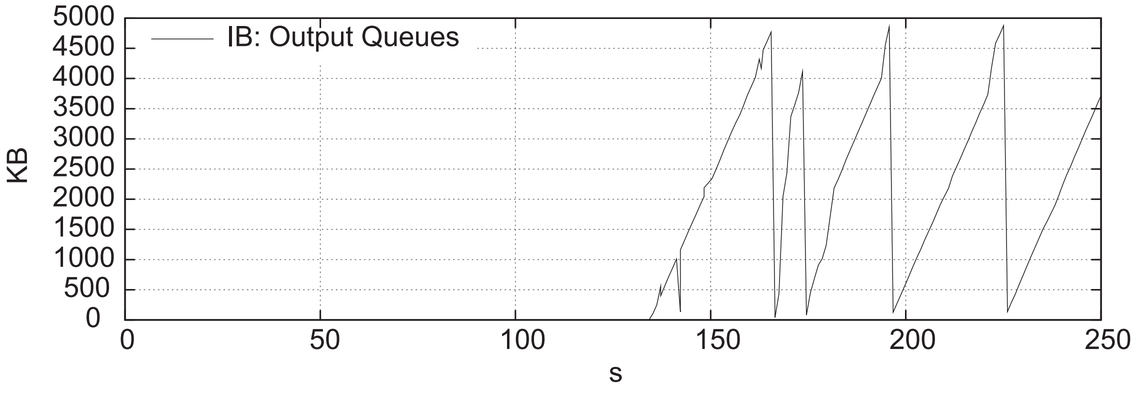


Figure 6.3: Queue utilisation at event broker IB during congestion collapse

messages, which creates congestion because the bandwidth bottleneck cannot accommodate the higher data rate due to retransmitted messages. As a result, the message rates observed at both subscriber-hosting brokers fall below the publication rate and never successfully recover.

Note that even though connections between event brokers use TCP congestion control, this is not sufficient to prevent congestion in the overlay broker network because of application-level queueing at event brokers. Both network and event broker congestion manifest themselves as the build up of buffer queues at event brokers. To deal with congestion, current middleware deployments are often vastly overprovisioned, which is a waste of resources. Instead, we propose a congestion control service for an event-based middleware to address the problem directly.

6.2 Requirements

A congestion control mechanism in a publish/subscribe context differs from traditional congestion control found in other networking systems. This is due to the many-to-many communication semantics supported by the publish/subscribe model and the content-based filtering of messages at application-level event brokers during event dissemination. Not all event subscribers receive the same set of publication messages sent by event publishers, as opposed to the case in application-level multicast, for example. Reliable event dissemination semantics leads to the selective retransmission of publication messages to a subset of recovering event subscribers, which

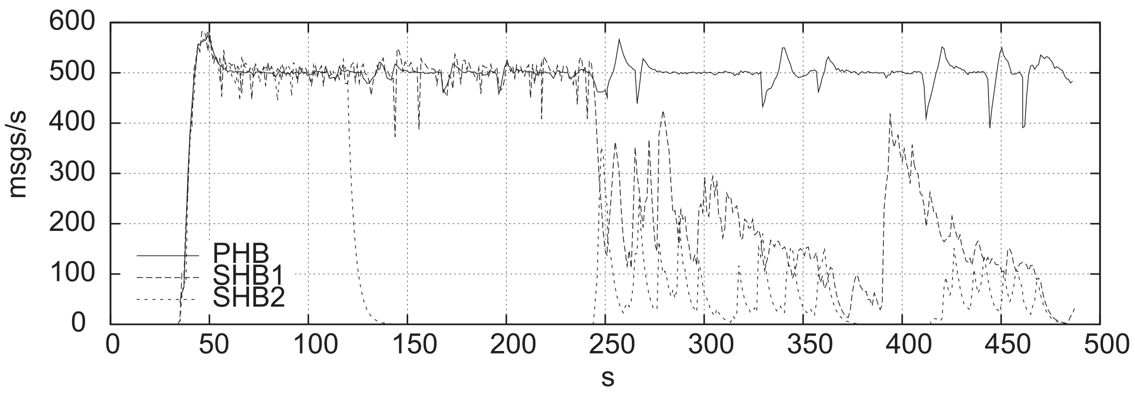


Figure 6.4: Congestion collapse with the PHB–IB link restricted during recovery

further complicates congestion control. To guide the design of our congestion control mechanism, we formulate six requirements for congestion control in an event-based middleware.

Burstiness. The processing of publication messages at event brokers is bursty because of application-level scheduling and the variable processing cost of content-based filtering of event publications. This means that a congestion condition can arise quickly, requiring early detection by the congestion control mechanism.

Queue Sizes. Due to the burstiness of event routing and the need to cache event streams for retransmission, buffer sizes at event brokers are much higher compared to standard networking components. Buffer overflow only occurs when significant congestion already exists in the system. As a consequence, message loss cannot be used as an indicator for congestion in the event-based middleware.

Recovery Control. The congestion control mechanism must ensure that event brokers that are recovering event publications that were previously lost will eventually complete recovery successfully. At the same time, recovering event brokers must be prevented from contributing to congestion. Although NACK messages are small and themselves cause little congestion, they potentially trigger the retransmission of large event publication messages.

Robustness. It is important that the congestion control mechanism is robust and can protect itself against malicious event clients. A possible design choice is to provide congestion control in the overlay broker network only, ensuring that the publication rate of messages by publisher-hosting brokers can be supported by all interested subscriber-hosting brokers. Flow control between client-hosting brokers and event clients is handled by a separate mechanism that can disconnect malicious clients.

Architecture Independence. The congestion control mechanism should not be tightly coupled to internal implementation details of an event broker. Instead, as a higher-level middleware service, it should support the evolution of the event broker implementation. For example, the detection of congestion should not depend on a particular buffer implementation or queueing discipline used by event brokers.

Fairness. When congestion requires the reduction of publication rates, fair throttling of event publishers must be ensured. The available resources at publisher-hosting brokers should be split equally among all hosted event publishers or pubends.

6.3 Congestion Control Algorithms

Typically a congestion control mechanism first detects congestion in the system and then adapts system parameters to remove its cause. In this section we describe two such algorithms that provide congestion control for an event-based middleware in accordance with the requirements stated in the previous section.

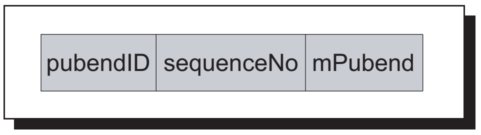
1. A *PHB-driven congestion control algorithm* ensures that publisher-hosting brokers cannot cause congestion because of too high a publication rate. This is achieved by a feedback loop between pubends and subscriber-hosting brokers to monitor congestion in the overlay broker network and control the event publication rate at the pubends.
2. An *SHB-driven congestion control algorithm* manages the recovery of subscriber-hosting brokers after failure. It limits the rate of NACK messages that cause the retransmission of event publications from pubends depending on congestion.

These two congestion control algorithms are independent of each other but should be used in conjunction to prevent congestion during both regular operation and recovery. Both algorithms need to distinguish between recovering and non-recovering event brokers in order to ensure that subscriber-hosting brokers can recover successfully. For a simpler presentation of the algorithms, we assume that only intermediate brokers are internal nodes in event dissemination trees with client-hosting brokers constituting the root or leaf nodes. Next we will describe the two algorithms in turn.

6.3.1 PHB-Driven Congestion Control

The *PHB-driven congestion control algorithm* (PDCC) controls the rate at which new publication messages are published by a pubend. The publication rate is adjusted depending on a *congestion metric*. We use the observed rate of publication messages at subscriber-hosting brokers as our congestion metric, which is similar to the throughput-based metric of TCP Vegas [BOP94]. The rationale behind this is that a decrease in the message rate at a subscriber-hosting broker with an unchanged publication rate at the pubend is an indication of more queueing in the overlay broker network. This queue build-up is considered to be caused by network or event broker congestion in the system. Subscriber-hosting brokers calculate their own congestion metric and notify the pubend whenever they believe that they are suffering from congestion. Congestion indications are aggregated at intermediate brokers so that the pubend is only informed of the worst congestion point. Two types of control messages are used to exchange congestion information between event brokers in an aggregated fashion.

Downstream Congestion Query (DCQ) Messages. The PDCC mechanism is triggered by DCQ messages sent by a pubend down the event dissemination tree to all subscriber-hosting brokers. Since congestion control is performed per pubend, a DCQ message carries a pubend identifier (`pubendID`). A monotonically increasing `sequenceNo` is used for aggregation and the `mPubend` field stores the current position in the pubend's event stream, which is for example the latest assigned event timestamp.



Upstream Congestion Alert (UCA) Messages. UCA messages are sent by subscriber-hosting brokers to inform a pubend about congestion. They flow upwards in the event dissemination tree and are aggregated at intermediate brokers so that a pubend only receives a single UCA message in response to a DCQ message. Apart from the pubend identifier and the sequence number of the triggering DCQ message, a UCA message contains the minimum throughput rates observed at recovering (`minRecSHBRate`) and non-recovering (`minNonRecSHBRate`) subscriber-hosting brokers.

pubendID	sequenceNo	minRecSHBRate	minNonRecSHBRate

Figure 6.5 summarises the propagation of DCQ and UCA messages through an overlay broker network in the PHB-driven congestion control algorithm. For the PDCC scheme to be efficient, DCQ and UCA messages must not suffer from congestion and should maintain low delays and loss rates. This can be achieved by treating them as high-priority control messages, as will be explained in Section 6.4. In the following we describe the behaviour of the three types of event brokers when processing DCQ and UCA messages in the PDCC algorithm.

Publisher-Hosting Broker (PHB). A publisher-hosting broker triggers the PDCC mechanism by periodically sending DCQ messages with an incremented sequence number. The interval t_{dcq} at which DCQ messages are dispatched determines the time between UCA responses in a congested system. The higher the rate of responses, the quicker the system adapts to congestion.

When the PHB has not received any UCA messages for a period of time t_{nouca}, it assumes that the system is currently not congested. Therefore, it increases the publication rate if the rate is throttled and the pubend could publish at a higher rate. To increase the publication rate, we use a hybrid scheme with additive and multiplicative increase. The new rate r_{new} is calculated from the old rate r_{old} according to

$$r_{\mathrm{new}} = \max\left[\; r_{\mathrm{old}} + r_{\mathrm{min}},\; r_{\mathrm{old}} + f_{\mathrm{incr}} * (r_{\mathrm{old}} - r_{\mathrm{decr}}) \;\right], \tag{6.1}$$

where r_{decr} is the publication rate after the last decrease, f_{incr} is a multiplicative increment factor, and r_{min} is the minimum possible increase. Initially we used a purely additive scheme but this resulted in a very slow increment and experiments showed that a more optimistic approach achieved a higher message throughput. The multiplicative use of f_{incr} allows the publication rate to grow faster than a fixed additive increase. However, when the publication rate is already close to the optimal operation point before congestion occurs, it is necessary to limit the increase.

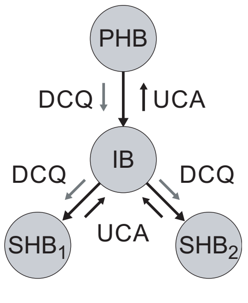


Figure 6.5: Flow of DCQ and UCA messages

This is done by recording the publication rate r_{decr} at which the increase started and using it to restrict the multiplicative increase. As will be shown by the experiments in Section 6.5, this scheme results in the publication rate probing whether the congestion condition has disappeared and, if not, oscillating around the optimal operation point.

When the PHB receives a UCA message, a decision is made about a reduction of the current publication rate. The rate is kept constant if the sequence number in the received UCA message is smaller than the sequence number of the DCQ message that was sent after the last decrease. The reason for this is that the system did not have enough time to adapt to the last change in rate and therefore more time should pass before another adjustment. The rate is also not reduced if the congestion metric in the UCA message is larger than the value in the previous message. This means that the congestion situation in the system is improving, and further reduction of the rate is unnecessary. Otherwise, the publication rate is decreased according to

$$r_{\text{new}} = \max \left[\, f_{\text{decr}_1} * r_{\text{old}},\ r_{\text{decr}} + f_{\text{decr}_2} * (r_{\text{old}} - r_{\text{decr}})\, \right] \quad \text{iff} \quad r_{\text{decr}} \neq r_{\text{old}} \tag{6.2}$$

$$r_{\text{new}} = f_{\text{decr}_1} * r_{\text{old}} \quad \text{otherwise,} \tag{6.3}$$

where f_{decr_1} and f_{decr_2} are multiplicative decrement factors. The first term in Equation 6.2 multiplicatively decreases the rate by a factor f_{decr_1}, whereas the second term reduces the rate relative to the previous decrement r_{decr}. Similar to Equation 6.1, the second term prevents an aggressive rate reduction when congestion is encountered for the first time after an increase. Since the PDCC mechanism constantly attempts to increase the publication rate in order to achieve a higher throughput, it will eventually cause SHBs to send UCA messages if there is resource shortage in the system, but this should not result in a strong reduction of the publication rate. Taking the maximum of the two decrement values ensures that the publication rate stays close to the optimal operating point. If the congestion situation does not improve after one reduction, the publication rate is reduced again. This time a strong multiplicative decrease according to Equation 6.3 is performed because the condition $r_{\text{decr}} = r_{\text{old}}$ holds.

Intermediate Broker (IB). To avoid the problem of feedback implosion [Dan89], aggregation logic for UCA messages at intermediate brokers must consolidate multiple messages from different SHBs such that the minimum observed rate at any SHB is passed upstream in a UCA message. This enables the pubend to adjust its publication rate to provide for the most congested SHB in the system. Another requirement is that UCA messages that occur for the first time are immediately sent upstream, allowing the pubend to respond as quickly as possible to new congestion in the system.

In Figure 6.6 the algorithm for processing DCQ and UCA messages at an intermediate broker is given. An IB stores the maximum sequence number `seqNo` and the minimum throughput values for non-recovering (`minNonRecSHBRate`) and recovering (`minRecSHBRate`) SHBs from the UCA messages that it has processed. After the initialisation of these variables (line 1), the function `processDCQ` handles DCQ messages by relaying them down the event dissemination tree in line 6. When a UCA message arrives, the function `processUCAMsg` is called, which first updates the throughput minima (lines 9–10). A new UCA message is only sent upstream if the sequence number of the received message is greater than the maximum sequence number stored at the IB (line 11). This ensures that UCA messages with the same sequence number coming from different SHBs are aggregated before propagation. The first UCA message with a new sequence number immediately triggers a UCA message so that the pubend is quickly informed about new congestion. Subsequent UCA messages from other SHBs that have the

```
1  initialization:
2    seqNo ← 0
3    minNonRecSHBRate ← ∞, minRecSHBRate ← ∞
4
5  processDCQ(dcqMsg):
6    sendDownstream(dcqMsg)
7
8  processUCA(ucaMsg):
9    minNonRecSHBRate ← MIN(minNonRecSHBRate, ucaMsg.minNonRecSHBRate)
10   minRecSHBRate ← MIN(minRecSHBRate, ucaMsg.minRecSHBRate)
11   IF ucaMsg.seqNo > seqNo THEN
12     sendUpstream(ucaMsg.seqNo, minNonRecSHBRate, minRecSHBRate)
13     seqNo ← ucaMsg.seqNo
14     minNonRecSHBRate ← ∞, minRecSHBRate ← ∞
```

Figure 6.6: Processing of DCQ and UCA messages at IBs

same sequence number will be aggregated and contribute towards the throughput minima in the next UCA message. After a UCA message has been sent in line 12, `seqNo` is updated (line 13) and both throughput minima are reset in line 14.

We demonstrate the operation of the aggregation logic at IBs with the example in Figure 6.7. The topology of six event brokers has two congested event brokers, SHB$_1$ and SHB$_2$, and three intermediate brokers IB$_{1,2,3}$ that aggregate UCA messages. Congestion in the system is first detected by SHB$_1$ and its UCA message with a congestion metric of 0.8 is directly propagated to the PHB. When SHB$_2$ notices congestion, its UCA message is consolidated at IB$_2$, which updates its throughput minimum to 0.4. Eventually a UCA message with the congestion value of SHB$_2$ will propagate up the event dissemination tree in response to a new DCQ message because SHB$_2$ is worse congested than SHB$_1$.

Subscriber-Hosting Broker (SHB). The congestion metric used by subscriber-hosting brokers depends on their observed throughput of publication messages and is independent of the actual publication rate of the pubend. An SHB monitors the ratio of pubend and SHB message rate,

$$t = \frac{r_{\text{pubend}}}{r_{\text{SHB}}}, \tag{6.4}$$

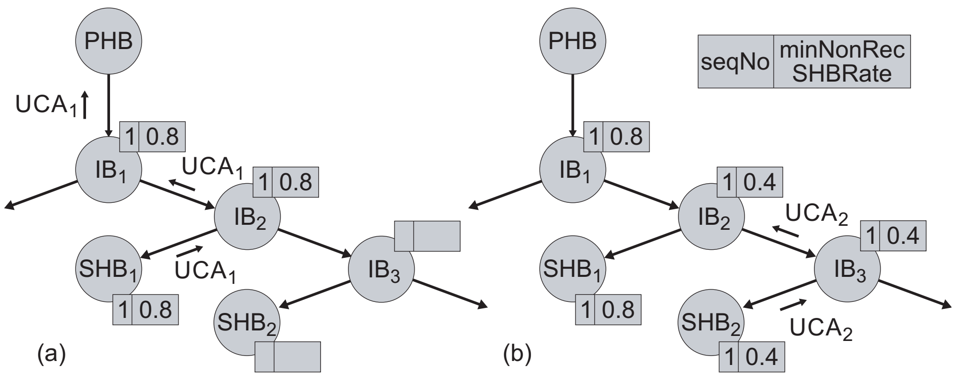


Figure 6.7: Consolidation of UCA messages at IBs

and uses this to decide when to send UCA messages with congestion alerts. To allow for burstiness in the throughput due to application-level routing as mentioned previously, t is passed through a standard first-order low pass filter,

$$\bar{t} = (1 - \alpha)\,\bar{t} + \alpha\, t, \tag{6.5}$$

to obtain a smoothed congestion metric $\bar{t}$ with an empirical value of $\alpha = 0.1$. An SHB has to apply a different strategy for sending UCA messages depending on whether it is recovering event publications or not. We assume that an SHB can determine whether it is a recovering or a non-recovering event broker. A suitable criterion to detect recovery would be, for example, that the SHB is ignoring new event publications because its event stream is saturated with old events caused by NACK messages.

Non-Recovering SHB. A non-recovering SHB should receive publication messages at the same rate at which they are sent by the pubend. Therefore, if the smoothed throughput ratio $\bar{t}$ drops below unity by a threshold Δt_{nonrec},

$$\bar{t} < 1 - \Delta t_{\text{nonrec}}, \tag{6.6}$$

the SHB assumes that it has started falling behind in the event stream because of congestion. In rare cases, an SHB could be falling behind slowly because $\bar{t}$ stays below 1 but above $1 - \Delta t_{\text{nonrec}}$ for a long time. Unless there is already significant congestion in the system, this will not cause a queue overflow if buffer sizes are large. An SHB can detect this situation by periodically comparing its current position in its event stream m_{SHB} to the pubend's event stream position m_{pubend} from the last received DCQ message. If the difference is larger than Δt_s,

$$m_{\text{SHB}} < m_{\text{pubend}} + \Delta t_s, \tag{6.7}$$

a UCA message is triggered, even though the congestion metric $\bar{t}$ is above its threshold value.

Recovering SHB. A recovering SHB must receive publication messages at a higher rate than the publication rate, or it will never manage to successfully catch up and recover all lost publication messages. In some applications there is an additional requirement to maintain a minimum recovery rate $1 + \Delta t_{\text{rec}}$ in order to put a bound on recovery time. Thus, a recovering SHB sends a UCA message if

$$\bar{t} < 1 + \Delta t_{\text{rec}}. \tag{6.8}$$

The threshold value Δt_{rec} influences how much of the congested resource will be used for recovery messages as opposed to new publication messages and hence controls the duration of recovery.

6.3.2 SHB-Driven Congestion Control

The *SHB-driven congestion control algorithm* (SDCC) manages the rate at which an SHB requests missed event publications by sending NACK messages upstream to the corresponding pubend. An SHB maintains a *NACK window* to decide which parts of the event stream to

Tick State	Description
Data	the tick contains an event publication
Silence	no message was published
Final	the tick is no longer needed
Question	the state of the tick is unknown

Table 6.1: The states of a tick in a Gryphon event stream

request. To control the rate of NACK messages being sent, the NACK window is open and closed additively by the SDCC algorithm depending on the level of congestion in the system. As for the PDCC mechanism, the change in recovery rate throughput is used as a metric for detecting congestion.

At the start of recovery, an SHB uses a small initial NACK window size $nwnd_0$. The NACK window is adjusted during recovery when the recovery rate r_{SHB} changes. The recovery rate r_{SHB} is defined as the ratio between the current NACK window size $nwnd$ and the estimate of the round trip time RTT, which it takes to retrieve a lost event publication from the pubend,

$$r_{\text{SHB}} = \frac{nwnd}{RTT}. \tag{6.9}$$

The NACK window size is changed in a similar fashion to TCP Vegas. When the recovery rate r_{SHB} increases by at least a factor α_{nack}, the NACK window is opened by one additional NACK message per round trip time. When r_{SHB} decreases by at least a factor β_{nack}, the NACK window is reduced by one NACK message,

$$nwnd_{\text{new}} = nwnd_{\text{old}} \pm size_{\text{nack}}. \tag{6.10}$$

This is sufficient to ensure that resent event publications triggered by NACK messages from recovering event brokers do not overload the event-based middleware.

6.4 Implementation

For evaluation with an industrial-strength, content-based publish/subscribe system, we have implemented the PDCC and SDCC algorithms as extensions to the guaranteed delivery service provided by the Gryphon event broker [BSB+02]. The implementation attempts to be independent of Gryphon-specific details so that it can be added as a higher-level service to any other event-based middleware. In this section we describe our prototype implementation starting with an explanation of the Gryphon guaranteed delivery service.

Under guaranteed delivery semantics in Gryphon, the event stream that is managed by event brokers is subdivided into discrete time intervals called *ticks*. Each tick potentially holds an event publication and is in one of four states listed in Table 6.1.

Ticks are fine-grained so that no two event publications are ever assigned to the same tick. The position of a tick in the event stream can be viewed as a real-time timestamp set by the pubend. When no events are published, ticks in the event stream are assigned to the silence state and the pubend sends an *explicit silence* message every t_{silence} ms to notify SHBs that they did not miss any event publications.

An example of an event stream maintained at a subscriber-hosting broker is shown in Figure 6.8. At first all ticks in the stream are initialised to the question state. The SHB then tries to resolve all question ticks to either data or silence states by sending NACK messages upstream to the pubend after a timeout if no event publication was received. When a tick has been successfully processed by an SHB, its state changes to final and it can be removed from the event stream. Every SHB maintains a *doubt horizon*, which is the position in the event stream until which there are no question ticks. All ticks with event publications before the doubt horizon either already were or can be delivered to event subscribers. In addition, the SHB keeps a *receive window* of all ticks that it is willing to process and a *NACK window* of ticks, for which retransmission can be requested. In our PDCC implementation, SHBs use the rate of progress of the doubt horizon r_{dh},

$$r_{\text{dh}} = \frac{\text{ticks}}{\text{time}} = t, \tag{6.11}$$

as the congestion metric t in order to detect congestion in the system. Since the event stream contains seconds worth of ticks, r_{dh} is has the dimension of "tick seconds" per second. The advantage of the doubt horizon is that its progress is independent of the actual publication rate of events at the pubend because the doubt horizon progresses constantly due to silence messages. Therefore it can be used as a throughput-based congestion metric in our congestion control algorithms.

As discussed earlier, the PDCC scheme relies on the free flow of DCQ and UCA messages through a congested system. The data path in a Gryphon event broker with all buffers is shown in Figure 6.9. The event broker has input and output buffer queues at the application-level and TCP buffers in the protocol stack implementation. DCA and UCA messages circumvent the regular data path taken by publication messages because they are handled in high priority input and output queues in the event broker. The fact that they share TCP buffers with other messages has little impact because the protocol buffers are much smaller than the application-level buffers.

6.5 Experiments

In this section we discuss four experiments E1–E4 that evaluate the PDCC and SDCC mechanisms under simulated congestion with two different overlay broker network topologies and show how congestion collapse is avoided. The setup for all experiments was a dedicated network of

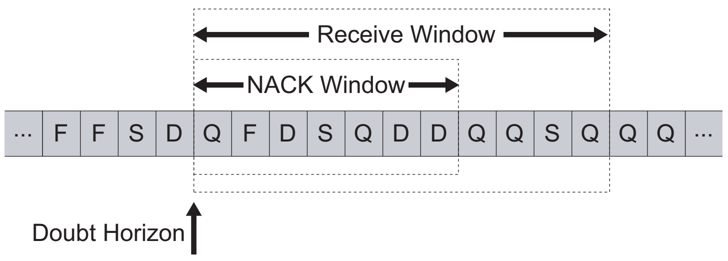


Figure 6.8: An event stream at an SHB

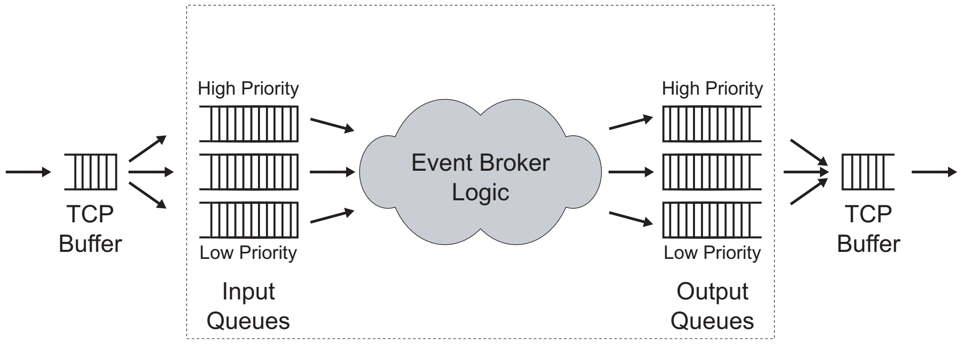


Figure 6.9: The data path for messages in a Gryphon event broker

Param.	**Description**	**Value**
t_{silence}	interval for sending explicit silence messages by pubend	1000 ms
b_{input}	size of input buffer queues at event broker	24 MB
b_{output}	size of output buffer queues at event broker	5 MB
t_{dcq}	interval for sending DCQ messages	1000 ms
t_{nouca}	interval without UCA messages before rate increase	2000 ms
r_{min}	minimum rate increase for PDCC scheme	2 msgs
f_{incr}	multiplicative increment for PDCC scheme	0.05
f_{decr_1}	multiplicative decrement for PDCC scheme	0.5
f_{decr_2}	multiplicative decrement w.r.t. previous increment	0.25
α	smoothing factor for low pass filter of congestion metric	0.1
Δt_{nonrec}	threshold value for UCA messages for non-recovering SHBs	50 tickms
Δt_{rec}	threshold value for UCA messages for recovering SHBs	1000 tickms
Δt_s	threshold value for lag in message stream for SHBs	4000 tickms
$nwnd_0$	initial size of NACK window	100 tickms
α_{nack}	recovery rate increase before NACK window is increased	0.1
β_{nack}	recovery rate decrease before NACK window is decreased	0.3

Table 6.2: Configuration parameters for Gryphon and the PDCC and SDCC algorithms

Gryphon event broker machines running AIX connected via Ethernet links. A summary of the configuration parameters, as described in the previous section, for Gryphon and the congestion control algorithms that were used in the experiments is given in Table 6.2. The failure of physical links was simulated by flushing an event broker's output buffers and closing its TCP connection. Network congestion was created as bandwidth limits on physical links by modifying an event broker's input buffers to only support a reduced processing rate. During an experiment run, various event broker statistics were recorded to create the plots in the following section.

6.5.1 E1: Link Failure

The first experiment E1 shows how the congestion control mechanisms handle congestion due to recovery after link failure. It is a re-run of the failure experiment from Figure 6.4 and uses the same topology. However, now the PDCC and SDCC schemes ensure that the system recovers successfully, as shown in the message rate plot in Figure 6.10. The publication rate of the PHB is reduced by the PDCC algorithm after the IB–SHB$_2$ link is reinstated at $t = 245$ because most

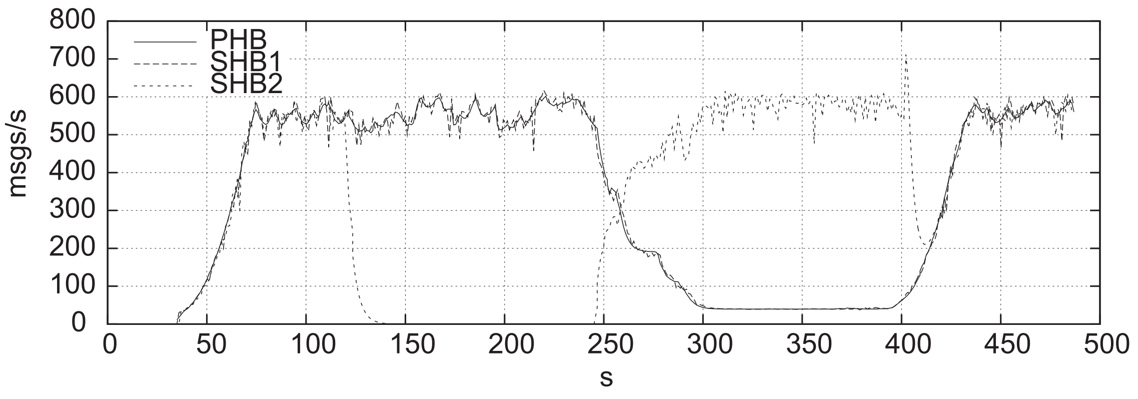


Figure 6.10: E1: Congestion control after an IB–SHB$_1$ link failure

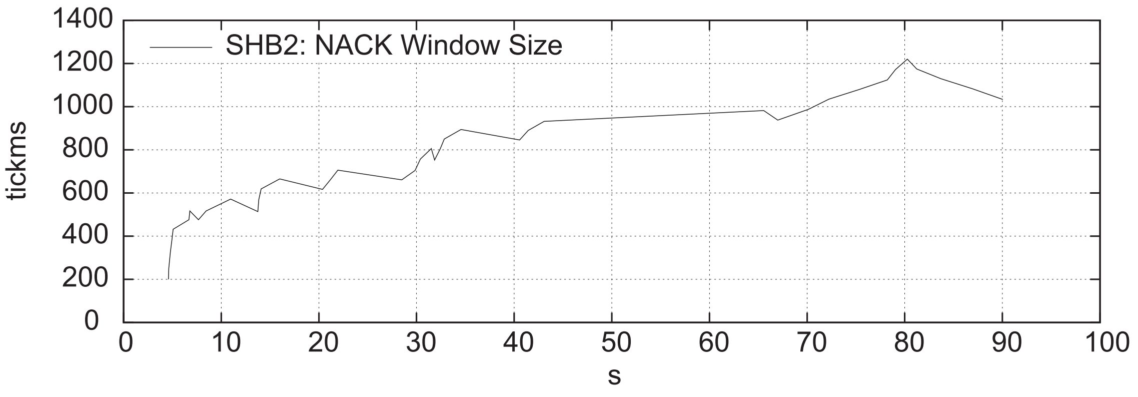


Figure 6.11: E1: NACK window behaviour after the IB–SHB$_1$ link failure

of its bandwidth is used by event broker SHB$_2$ for recovery. After SHB$_2$ has finished recovery at $t = 405$, the PHB restores its publication rate back to the previous maximum value. Note that the spike in SHB$_2$'s event rate close to the end of the recovery phase occurs because the IB caches recent ticks in the event stream and is therefore able to satisfy some of the final NACK messages more quickly.

The plot in Figure 6.11 presents the behaviour of the NACK window during recovery, as controlled by the SDCC algorithm. The NACK window starts with a small size of 200 tickms and is then progressively enlarged until an optimal operation point of around 900 tickms is reached. This controlled behaviour of the NACK window ensures that the system is not overwhelmed by retransmitted event publications. The NACK window increases further towards the end of the recovery phase because of the cached ticks at the IB.

6.5.2 E2: Bandwidth Limits

We investigated in experiment E2 how well the PDCC mechanism can adapt to dynamic bandwidth limitations similar to the one that caused congestion collapse in Figure 6.2. The observed message rates using the four event-broker topology from Figure 6.1 are shown in Figure 6.12. At first, the IB–SHB$_1$ link is restricted to about one third of the required bandwidth for 120 s. After that, the limit is increased to about half the necessary bandwidth for 120 s, and then reduced to the previous low value again. As can be seen from the publication rate graph, the

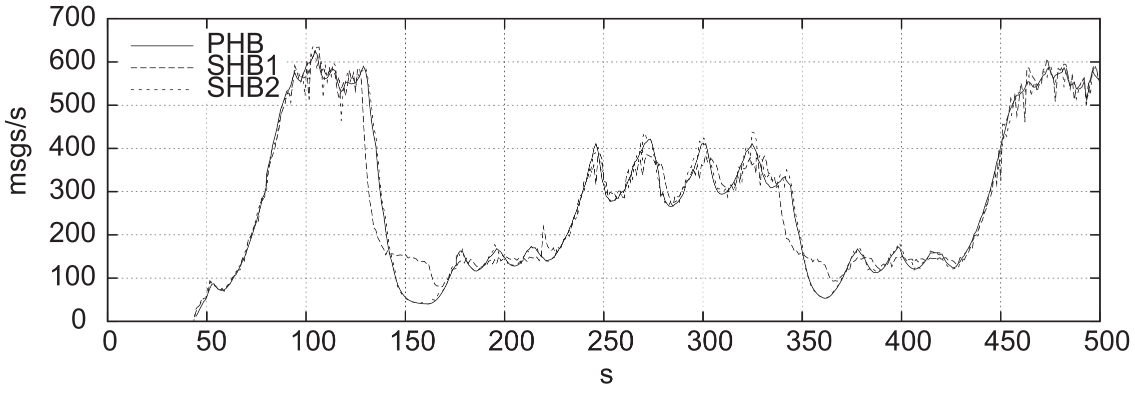


Figure 6.12: E2: Congestion control with dynamic bandwidth restrictions

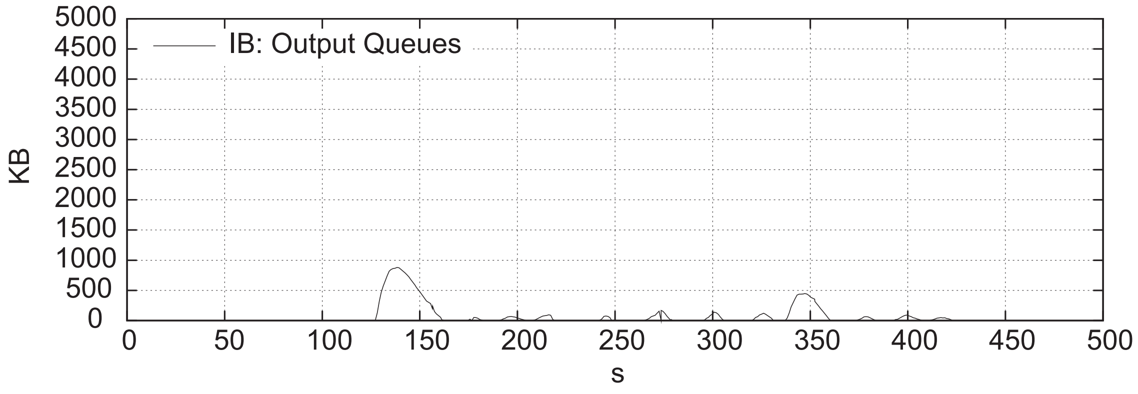


Figure 6.13: E2: Output queue utilisation at broker IB

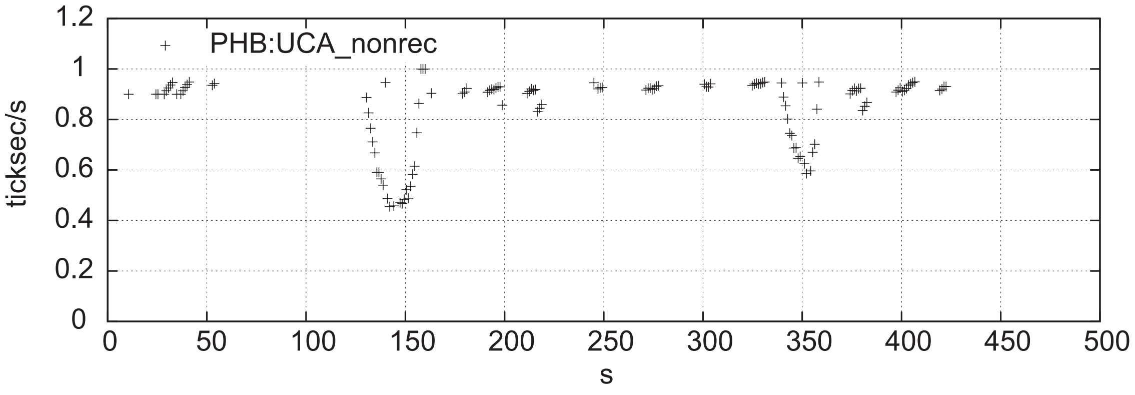


Figure 6.14: E2: UCA messages received at pubend

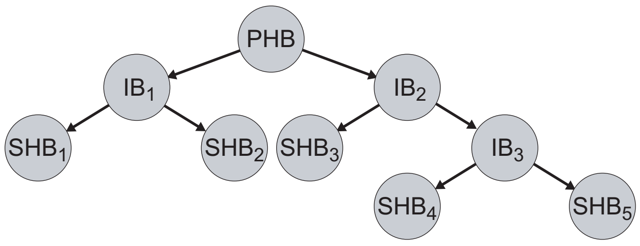


Figure 6.15: A complex overlay broker network topology with sixteen event brokers

PDCC algorithm determines the optimal rate that can be supported by the restricted link without causing congestion and manages to quickly adapt to new bandwidth bottlenecks. While adapting, the output buffer utilisation at the IB, which is shown in Figure 6.13, is kept low. Even when the available bandwidth is severely restricted at $t = 100$ and $t = 340$, the PDCC algorithm is responsive enough to prevent queues at the IB from increasing above 1 MB. The publication rate oscillates around the optimal value since the PHB is constantly probing the system to determine whether the congestion situation has improved. The r_{decr} mechanism in the PDCC algorithm ensures that the publication rate stays close to the optimal value.

To illustrate the operation of the PDCC algorithm, Figure 6.14 plots the doubt horizon rate r_{dh} from UCA messages received at the pubend. When there is no congestion during the first 120 s, UCA messages are not received except for transient messages at start-up. At $t = 140$ the doubt horizon rate decreases to half of its previous value because of the new link bottleneck. Once the publication rate at the pubend has been reduced sufficiently, the doubt horizon rate starts increasing again. When the link bottleneck is unchanged, UCA messages with doubt horizon rates slightly below the "real-time" rate of 1 ticksec/s are received periodically, preventing the publication rate from increasing further.

6.5.3 E3: Link Failures and Bandwidth Limits

To evaluate how multiple sources of congestion in different parts of the network are handled, we set up a complex overlay broker network for experiment E3. This topology has one publisher-hosting broker PHB, three intermediate brokers IB_{1-3}, and five subscriber-hosting brokers SHB_{1-5}, interconnected as shown in Figure 6.15. The topology is asymmetric with different path lengths from SHBs to the PHB to ensure that IBs have to perform non-trivial aggregation of UCA messages. Note that due to the distribution of subscribers in this experiment, the event rate at an SHB should be 2/5th of the PHB's publication rate under normal conditions.

The experiment consists of a series of link failures and bandwidth limitations. Throughout the entire experiment, the IB_1–SHB_1 link has a bandwidth limitation that does not cause congestion in the absence of failure. At $t = 190$, the PHB–IB_1 link is limited to half the required bandwidth, and, after a further 120 s, the IB_3–SHB_5 link is failed for 120 s ($t = 310 \ldots 430$). The event rates at the PHB and the SHBs are plotted in Figure 6.16. At the beginning, the PHB publishes events at the maximum rate but when the first bandwidth restriction comes into effect, all SHBs receive events at a reduced rate because of the PDCC rate adjustment. After the failure of the IB_3–SHB_5 link, the PHB decreases its rate even further to enable SHB_5 to recover all lost event publications. Even though failure and link restrictions occur in different parts of the network at the same time, the PDCC algorithm drives the publication rate at the pubend by the behaviour of the worst congestion point and successfully prevents queues from building up.

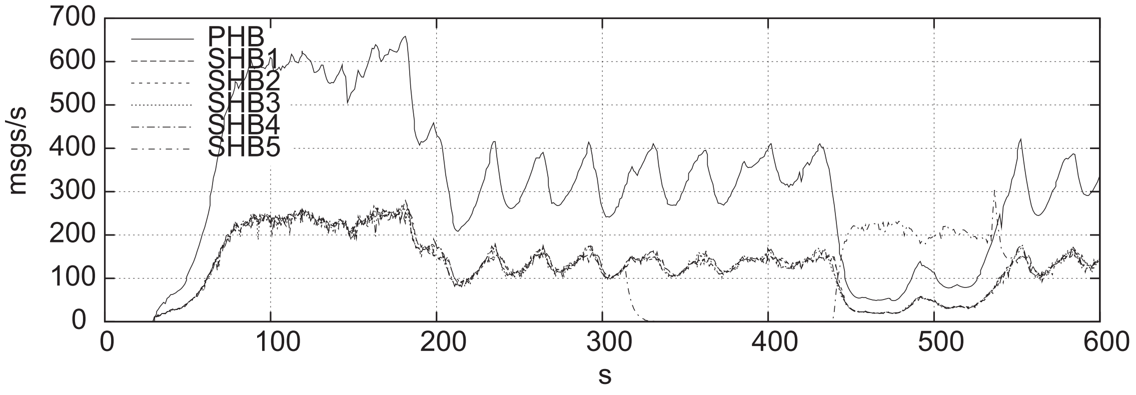


Figure 6.16: E3: Congestion control with link failures and bandwidth restrictions

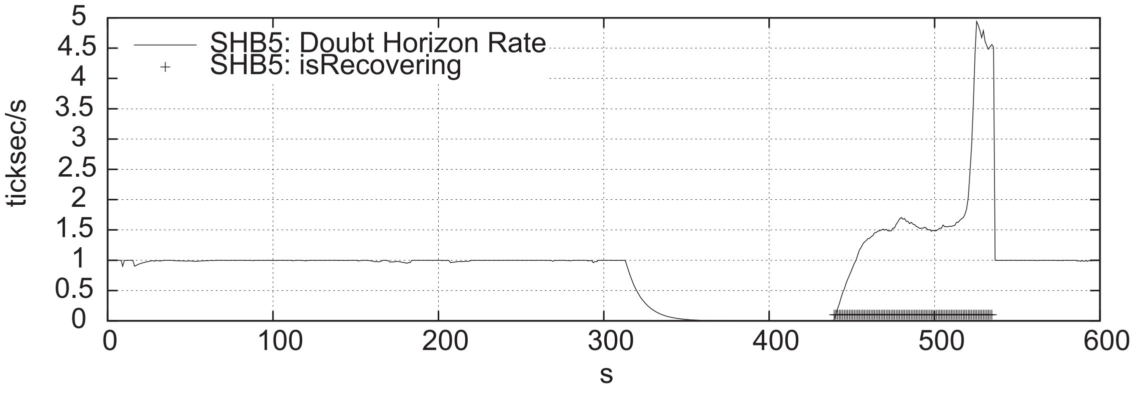


Figure 6.17: E3: Doubt horizon rate with link failures and bandwidth restrictions

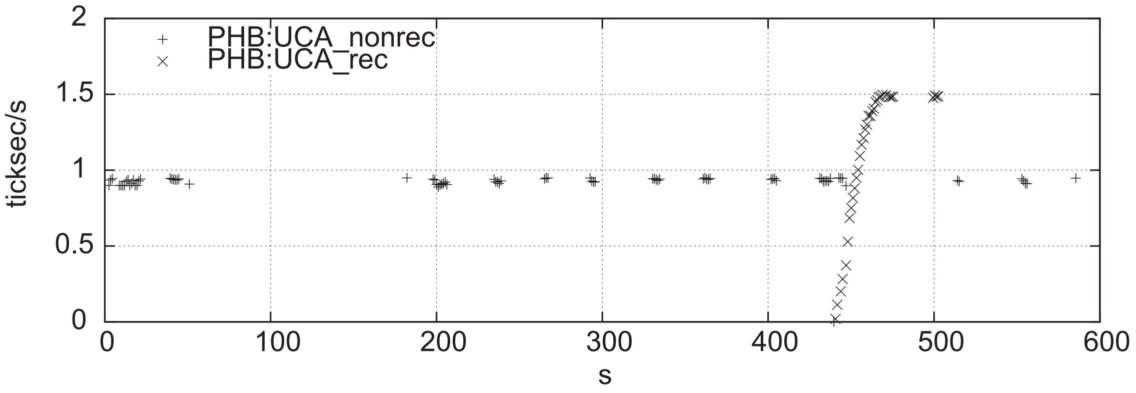


Figure 6.18: E3: UCA messages received at the pubend

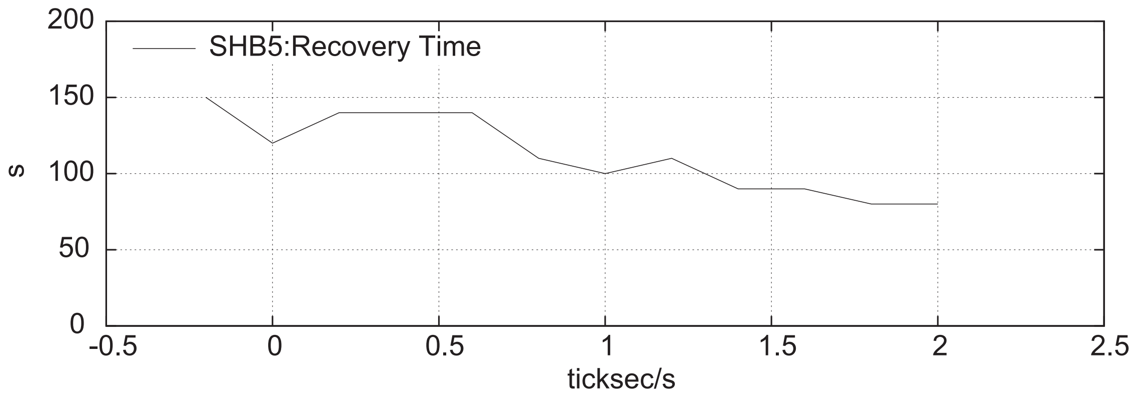


Figure 6.19: E4: Variation of recovery time with the Δt_{rec} threshold

The doubt horizon rate r_{dh}, as observed at SHB_5, can be seen in Figure 6.17. Since the rate is independent of the publication rate, it stays close to 1 ticksec/s until the IB_3–SHB_5 link is failed at $t = 310$. The value of Δt_{rec} in this experiment is 0.5 ticksec/s. After the failure, the SHB switches to recovery mode (isRecovering) and the doubt horizon rate is kept above Δt_{rec}. Close to the end, the rate peaks to about 5 ticksec/s when the SHB_5 reaches the point in the event stream when the pubend reduced its rate and thus more ticks in the stream are small silence ticks without data.

Figure 6.18 shows the consolidated UCA messages received at the PHB from recovering and non-recovering SHBs. After a startup effect, messages from non-recovering SHBs ($\text{UCA}_{\text{nonrec}}$) arrive at regular intervals because of the PHB–IB_1 link restriction. When SHB_5 starts recovering at $t = 430$, it sends UCA messages in recovery mode (UCA_{rec}) whenever its rate drops below Δt_{rec}.

6.5.4 E4: Recovery Times

The final experiment E4 investigates how the duration of recovery can be influenced by the threshold value Δt_{rec}. In general, the higher this value, the earlier a recovering SHB will send UCA messages so that more of the congested resource is used for retransmitted event publications as opposed to new ones. This experiment uses the four event-broker topology with a bandwidth restriction on the PHB–IB link and a failure of the IB–SHB_2 link. Figure 6.19 plots the recovery time in seconds as a function of Δt_{rec} varying from $-0.2 \ldots 2.0$ ticksec/s at 0.2 ticksec/s steps. A value of $\Delta t_{\text{rec}} \leq 0$ is not used in practice, as it may result in an infinite recovery time.

The plot shows a clear correlation between Δt_{rec} and the recovery duration. However, it is not linear for two reasons: With $\Delta t_{\text{rec}} \geq 1.8$, the bandwidth restricted PHB–IB link becomes saturated with resent publication messages. When $\Delta t_{\text{rec}} \leq 1$, there is an interaction between the four pubends hosted at the PHB. Although the low threshold value does not force the pubends to reduce their publication rate by much, some pubends tend to consume a larger fraction of the bottleneck bandwidth, reducing the available bandwidth for the remaining pubends. These pubends observe a very low r_{dh} value and thus reduce their publication rate more than necessary. This unfairness between pubends stems from the first-come, first-serve scheduling of messages of the same priority at event brokers, which prevents the PDCC algorithm from synchronising rate reductions at multiple pubends.

6.6 Related Work

A large body of work exists in the area of congestion control in networks, although these solutions do not address the special requirements for congestion control in an event-based middleware (see Section 6.2). In this section we provide a brief overview of applicable work, contrasting it with our approach for congestion control.

Transmission Control Protocol (TCP). The TCP protocol comes with a point-to-point, end-to-end congestion control algorithm with a congestion window that uses *additive increase, multiplicative decrease* (AIMD) [JK88]. *Slow start* helps open the congestion window more quickly. Packet loss is the only indicator for congestion in the system and *fast retransmit* enables the receiver to signal packet loss by ACK repetition to avoid timeouts. TCP Vegas [BOP94] attempts to detect congestion before packet loss occurs by using a throughput-based congestion metric, which is similar to the congestion metric used in the PDCC and SDCC algorithms.

Reliable Multicast. Reliable multicast protocols are similar to reliable publish/subscribe systems due to their one-to-many communication semantics, but typically they have no filtering at intermediate nodes and do not guarantee that all leaves in the multicast tree will eventually catch up with the sender. In general, multicast congestion control schemes can be divided into two categories [YL00], namely:

1. *sender-based* schemes, in which all receivers support the same message rate, and
2. *receiver-based* schemes with different message rates by means of transcoded versions of data.

Since we can make few assumptions about the content of event publications, a receiver-based approach is not feasible. Congestion control for multicast is often implemented at the transport level relying on router support. It must adhere to existing standards to ensure fairness and compatibility with TCP [FF99, GS99]. Since there are many receivers in the multicast tree, scalable feedback processing of congestion information is important. Unlike *feedback suppression* [DO97], our approach does not discard information because it consolidates feedback in a scalable way.

The *pgmcc* congestion control protocol [Riz00] forms a feedback loop between the sender and the worst congested receiver. The sender chooses this receiver depending on receiver reports in NACK messages. The congestion control protocol for *SRM* [SW00] is similar except that the feedback agent can give positive and negative feedback, and a receiver locally decides whether to send a congestion notification upstream to compete for becoming the new feedback agent. An approach that does not rely on network support, except minimal congestion feedback in NACK messages, is *LE-SBCC* [TSLK01]. Here a cascaded filter model transforms the NACK messages from the multicast tree to appear like unicast NACKs before feeding them into an AIMD module. However, no consolidation of NACK messages can be performed. All these schemes have in common that they use a loss-based congestion metric, which is not a good indicator for congestion in an application-level overlay network.

Multicast Available Bit Rate (ABR) ATM. The ATM Forum Traffic Management Specification [Sat96] includes an Available Bit Rate (ABR) category for traffic though an ATM network. At connection setup, *forward and backward resource management* (FRM/BRM) cells are exchanged between the sender and receiver to create a resource reservation, which is modified at intermediate ATM switches. All involved parties agree on an acceptable cell rate depending on the congestion in the system. In our case, it is difficult to determine an acceptable message rate for an IB since the cost of processing event publications varies depending on size, content, and event subscriptions.

Multicast ABR requires flow control for one-to-many communication. An FRM cell is sent by the source and all receivers in the multicast tree respond with BRM cells, which are consolidated at ATM switches [Rob94]. Different ways of consolidating feedback cells have been proposed [FJG+98]. These algorithms have a trade-off between timely response to congestion and the introduction of *consolidation noise* when new BRM cells do not include feedback from all downstream branches. Our consolidation logic at intermediate brokers tries to balance this trade-off by aggregating UCA messages with the same sequence number, but also short-cutting new UCA messages. The scalable flow control protocol in [ZSSK02] follows a soft synchronisation approach, where BRM cells triggered by different FRM cells can be consolidated at a branch point.

Overlay Networks. Congestion control for application-level overlay networks is sparse, mainly because application-level routing is a novel research focus. A hybrid system for application-level reliable multicast in heterogeneous networks that addresses congestion control is *RMX* [CMB00]. It uses a receiver-based scheme with the transcoding of application data. Global flow control in an overlay network can be viewed as a dynamic optimisation problem [AADS02], in which a cost-benefit approach helps find an optimal solution.

6.7 Summary

In this chapter we have addressed the requirement for congestion control in an event-based middleware as a higher-level middleware service. We motivated the need for congestion control by showing two examples of congestion collapse due to resource shortage during normal operation and recovery. From this, we derived requirements for a congestion control mechanism focusing on the differences from traditional networking environments. Our solution consists of a PHB-driven congestion control (PDCC) algorithm that ensures that the publication rate of new messages can be supported by the middleware by exchanging DCQ and UCA messages for congestion notification. Feedback is consolidated at intermediate brokers in an efficient and scalable way. An SHB-driven congestion control (SDCC) algorithm regulates the rate of NACK messages during recovery in relation to the available resources in the system.

The prototype implementation of the algorithms in the Gryphon event broker was used to evaluate our solution within two different topologies. The four experiments showed that the presented schemes were capable of handling dynamic bandwidth limitations in the network, thus ensuring that congestion collapse could not occur. Moreover, additional resource demands because of recovery after failure are handled graciously and guarantees about recovery within a finite time are provided to the clients.

7 Composite Event Detection

For certain applications, the expressiveness of subscriptions in our event model from Section 3.3.1 is not sufficient. As a remedy, we propose a higher-level composite event service [PS02, PSB03, PSB04] that facilitates the management of a large volume of events by enabling event subscribers to specify their interest more precisely. The composite event service supports the advanced correlation of events through the detection of complex event patterns. In this chapter we describe our composite event service for an event-based middleware. It consists of composite event detectors, implemented as extended finite state automata, for the detection of composite events and a composite event language for their specification. To ensure scalability, the composite event service decomposes complex composite event subscriptions and automatically distributes the detectors throughout the overlay broker network in the event-based middleware.

We introduce the concept of a composite event in the next section. In Section 7.2 we motivate the need for composite event detection with three application scenarios. After that, the design and architecture of the composite event service is presented in Section 7.3. We describe the two main parts of the service: the composite event detection automata (Section 7.4) and the composite event language (Section 7.5). The distributed detection of composite events is discussed in Section 7.6. We conclude the chapter with an overview of our prototype implementation, including an experimental evaluation, in Section 7.7, and a discussion of related work in Section 7.8.

7.1 Composite Events

A composite event service is based on the notion of a *composite event.* Composite events prevent event subscribers from being overwhelmed by a large number of primitive event publications by providing them with a higher-level abstraction. A composite event is published whenever a certain pattern of events occurs in the event-based middleware. This enables event subscribers to directly subscribe to complex event patterns, as opposed to having to subscribe to all the primitive events that make up the pattern and then perform the detection themselves.

When a composite event has been detected by the composite event service, it is published and contains all events that contributed to its occurrence. Since our event-based middleware is strongly-typed, every composite event has a *composite event type* that is built from the event types of the included events and the relation between them in the composite event subscription. Note that primitive events can also be considered degenerate composite events, thus unifying primitive and composite event types within the middleware.

Definition 7.1 (Composite Event) *Every* composite event *c has a composite event type τ_c and belongs to the composite event space $\mathbb{C}$,*

$$(c : \tau_c) \in \mathbb{C}$$

A composite event type *τ_c corresponds to a valid expression $\mathcal{C}$ in a composite event language,*

$$\tau_c \equiv \mathcal{C}.$$

A composite event c consists of an interval timestamp t_c and a set of composite sub-events $\{c_1, c_2, \ldots, c_k\}$,

$$c : \tau_c = (t_c, \{c_1, c_2, \ldots, c_k\}).$$

A composite event is associated with a timestamp t_c that denotes when it has occurred. In a distributed system, there is no concept of global time [Lam78], which is why we assume partially-ordered *interval timestamps* [LCB99]. An interval timestamp has a start and end time so that it can express the local clock uncertainty at an event broker and also the duration associated with a composite event from the first contributing event to the last. We define a partial and a total order over interval timestamps that will be used by the strong and weak transitions in our detection automata described in Section 7.4.

Definition 7.2 (Interval Timestamp) *An* interval timestamp *t_c,*

$$t_c = [t_c^l; t_c^h],$$

has a start time t_c^l and an end time t_c^h with $t_c^l \leq t_c^h$. Interval timestamps are partially-ordered ($<$) and totally-ordered ($\prec$) as follows,

$$\begin{aligned} t_{c_1} < t_{c_2} &\triangleq t_{c_1}^h < t_{c_2}^l, \\ t_{c_1} \prec t_{c_2} &\triangleq (t_{c_1}^h < t_{c_2}^h) \vee (t_{c_1}^h = t_{c_2}^h \wedge t_{c_1}^l < t_{c_2}^l). \end{aligned}$$

A composite event language allows event subscribers to submit composite event subscriptions to the composite event service, which will then trigger the publication of composite events. Before presenting our composite event language, we survey possible application scenarios that benefit from composite event detection and hence obtain an understanding of the required features of a composite event language.

7.2 Application Scenarios

Many application scenarios for an event-based middleware benefit from a general purpose composite event service that enhances the expressiveness of the middleware. In the following we will consider how composite events can be used in a ubiquitous computing environment, for network systems monitoring, and for the dissemination of trust information in a large-scale distributed system. For each application scenario, we provide two examples of composite event subscriptions.

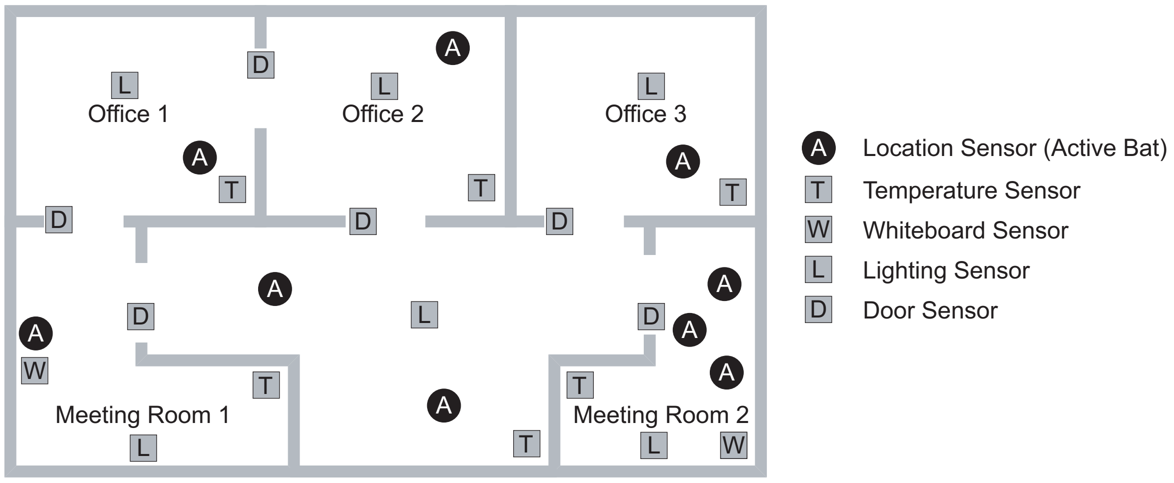


Figure 7.1: The Active Office with different sensors

The Active Office. The Active Office is a computerised building that was introduced in Section 1.2.2. Building users wear *Active Bats* [ACH$^+$01] that publish location information and static sensors gather data about doors, lighting, equipment usage, and environmental conditions, as shown in Figure 7.1. Composite event detection can help process the primitive events produced by the large number of sensors and provide a higher-level abstraction to users of the Active Office.

1. A user may subscribe to be notified when a meeting with at least three people working in the messaging department takes place during working hours in one of the meeting rooms.

2. Building services may be interested in composite events about a drop in temperature under 15 degrees for at least 15 min in any occupied office.

Network Systems Monitoring. When monitoring the operation of networks [BCF94], network entities publish events that are related to fault conditions in the network, such as shown in Figure 7.2. In practice, millions of events may be published daily in respect of fewer than a hundred real faults that require human intervention. The task of network systems monitoring

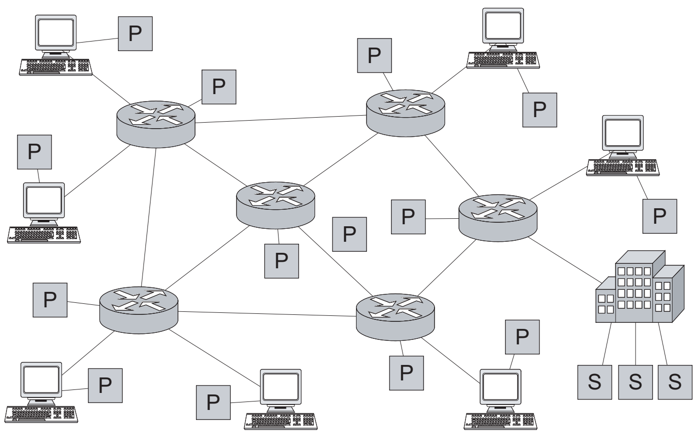


Figure 7.2: A system for monitoring faults in a network

can thus be simplified by expressing patterns associated with real problems as composite event subscriptions.

1. The network management centre may want to be notified when at least five workstations in different parts of the network detect a degradation in network bandwidth.
2. A network customer may be interested in composite events when none of its load-balanced web servers are available to the outside world unless the downtime is part of maintenance work.

The XenoTrust Framework. Recent efforts, such as the XenoServer project [BDF$^+$03], are building large-scale, public infrastructures for general-purpose, distributed computing with resource management and sharing. In such environments, reputation information about participants must be disseminated in a timely and scalable fashion so that entities can make trust-dependent decisions. Composite events can help participants receive notifications about changes in reputation of their resource providers or consumers, thus creating a global-scale trust management system [KDHP03].

1. A user may want to be notified when the reputation of any of its currently active resource providers drops below a certain threshold and there is an alternative provider that is capable of taking over the current resource contract.
2. A resource provider may submit a subscription causing a composite event when a new client receives a low reputation rating from at least three other providers within three days while requesting significant resources.

7.3 Design and Architecture

In this section the design and architecture of our composite event service is given. The design requirements for the service can be derived from the above application scenarios. In general the service should be applicable to a wide range of event-based middleware designs and therefore make few assumptions about the underlying publish/subscribe implementation. Ideally it should only rely on standard interfaces provided by the event-based middleware and not require special extensions to the event model. For example, content-based routing and filtering support should be exploited for the dissemination of composite events. To satisfy the requirement of scalability, composite event detection must be distributed so that complex composite event subscriptions can be decomposed into subexpressions that are then detected at different nodes in the system.

From the description of the application scenarios, it becomes clear that it is challenging to design a single composite event language that is both expressive and intuitive to use by, say, a human user in the Active Office environment. Therefore we propose the approach in Figure 7.3 with several specification layers that have different powers of expressiveness. At the bottom layer, *composite event detection automata* provide maximum expressiveness and perform the actual detection of composite events described in the next section. Composite event subscriptions specified in the *core composite event language* presented in Section 7.5 can be decomposed for distributed detection. Finally, domain-specific *higher-level languages* constitute the top layer and only expose a subset of the core language — supporting a simpler definition of composite events for a given application domain. Expressions in higher-level languages are automatically compiled down to composite event detection automata by the composite event service.

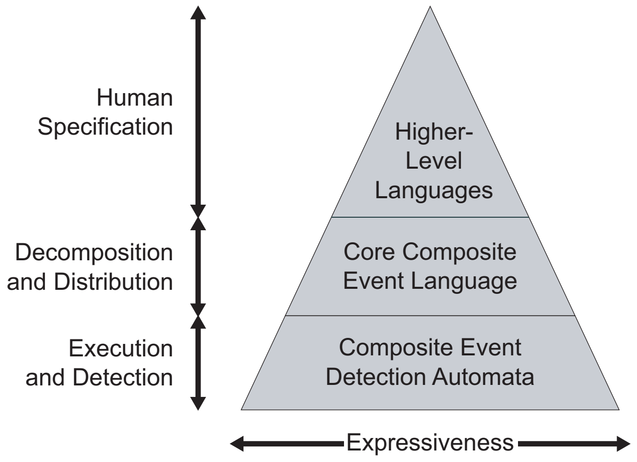

Human Specification
Decomposition and Distribution
Execution and Detection
Higher-Level Languages
Core Composite Event Language
Composite Event Detection Automata
Expressiveness

Figure 7.3: The components of the composite event detection service

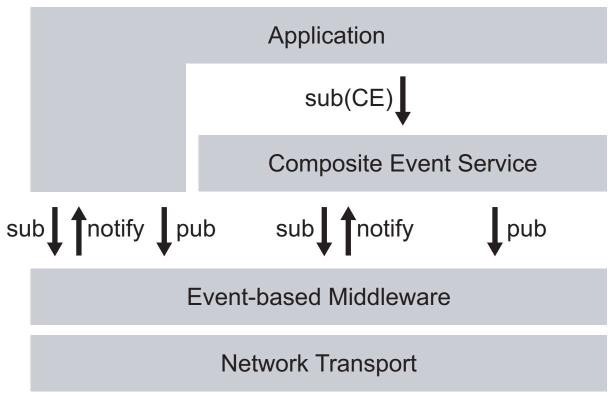


Figure 7.4: The architecture for the composite event detection service

The architecture for our composite event service is shown in Figure 7.4. The service uses the event client API of the event-based middleware so that composite event detectors can subscribe to primitive events and detect the occurrence of composite events. The event-based middleware is also used to coordinate the detection of decomposed composite event expressions and publish detected composite events. Note that the event-based middleware does not need to be aware of composite event types because composite event publications can be disguised using new primitive event types. Content-based routing and filtering of events is carried out by the event-based middleware. An application with event clients can either use the composite event service to submit composite event subscriptions and cause the instantiation of detectors, or interact directly with the event-based middleware for normal middleware functionality.

7.4 Composite Event Detection Automata

The composite event service uses *composite event detection automata* — which are finite state automata [HMU01] that are extended with support for temporal relationships and concurrent events — to analyse event streams. Basing composite event detection on extended finite state automata has several advantages. First, finite state automata are a well-understood computational model with a simple implementation. Their restricted expressive power has the benefit of limited, predictable resource usage, which is important for the safe distribution of detectors

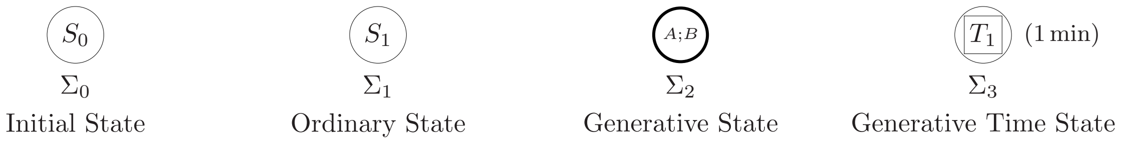


Figure 7.5: The states in a composite event detection automaton

in the event-based middleware. Regular expression languages have operators that are tailored towards the detection of patterns, which avoids the risk of redundancy or incompleteness when defining a new composite event language. In addition, complex expressions in a regular language may easily be decomposed for distributed detection.

A detection automaton consists of a finite number of states and transitions between them. To ensure that each state only has to consider certain events for transitions, it is associated with an *input domain* Σ, which is a generalisation of the concept of an input alphabet in traditional finite state automata. An input domain is a collection of *describable event sets* $A, B, C, \ldots$, which correspond to sets of events that are matched by a primitive or composite event subscriptions. In a given state, only these events need to be considered by the automaton because other events are not relevant for the composite event being detected. In practice, the automaton issues subscriptions for all describable event sets in the input domain of a state. The resulting incoming events are ordered according to the total timestamp order ($\prec$) (see Definition 7.2) into an *event input sequence* and are consumed by the automaton sequentially.

As shown in Figure 7.5, a detection automaton has four kinds of state. Detection starts in a unique *initial state* and continues through a series of *ordinary states*. A *generative state* is an accepting state that also publishes the composite event that has been detected by the automaton. *Generative time states* deal with timing by publishing an internal *time event* when a timer associated with the state expires. The automaton treats time events like regular events but they are not visible externally.

Each state can have two forms of outgoing transition that are labelled with the describable event sets of the events that trigger them. Note that since describable event sets can be defined by composite event subscriptions, our automata support the detection of event patterns involving concurrency. In Figure 7.6 the transition between states S_1 and S_2 is a *weak transition* that requires the timestamps of the events from the describable event sets A and B to be partially-ordered ($<$). A *strong transition*, such as between states S_2 and S_3, mandates a total ordering ($\prec$) between events from B and C. Strong and weak transitions therefore allow the expression of different temporal orderings between events. When an event that is part of the input domain but without a matching outgoing transition is received, the detection in the automaton fails. Several matching transitions and empty ϵ-transitions are followed nondeterministically. Although the following presentation of the detection automata uses nondeterminism, standard techniques can be used to convert them into deterministic automata [HMU01].

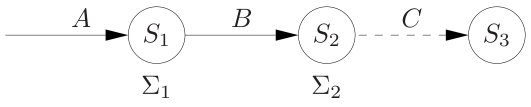


Figure 7.6: The transitions in a composite event detection automaton

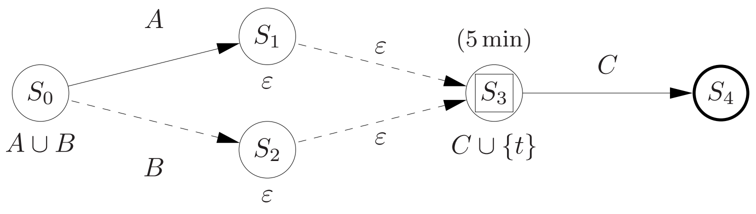


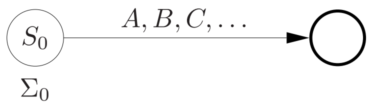
Figure 7.7: A composite event detection automaton

We illustrate our composite event detection automata with the example in Figure 7.7. This automaton starts in state S_0 with an input domain of $A \cup B$. A strongly-followed event from A causes a transition to state S_1, a weakly-followed event from B leads to state S_2. Once the generative time state S_3 is reached, a timer starts that will expire after 5 min, publishing time event t. Since this event is part of the input domain for state S_3 but there is no corresponding outgoing transition, detection will fail unless an event from C is received before the timer expires, triggering a transition to state S_4. The generative state S_4 signals the successful detection of a composite event with a composite event publication.

7.5 Composite Event Language

Composite event subscriptions use a *core composite event language* to specify the set of composite events that an event client is interested in. In this section we introduce the language operators and the construction of corresponding composite event detection automata from sub-automata. Some operators in our language, namely concatenation, alternation, and iteration, are influenced by those found in regular languages. However, other operators reflect the special features of our detection automata. After the description of the operators, we give examples for core language expressions and discuss three higher-level languages for composite event specification.

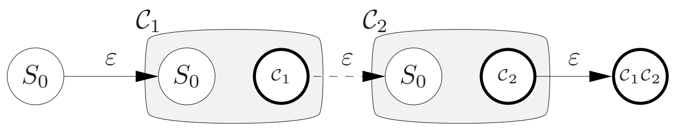
Atoms. $[A, B, C, \ldots \subseteq \Sigma_0]$.



Atoms detect individual events in the input stream of all events that are in the input domain Σ_0. Here only events in the describable event sets $A \cup B \cup C \cup \ldots$ are matched and cause a transition to a generative state. Other events in Σ_0 result in failed detection, and events outside Σ_0 are ignored. The trivial atom $[A \subseteq A]$ is abbreviated as $[A]$.

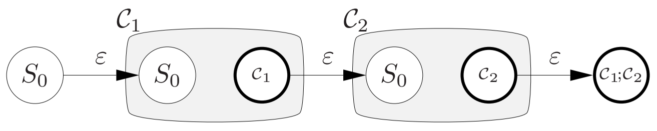
Negation. $[\neg E \subseteq \Sigma] \triangleq [\Sigma \backslash E \subseteq \Sigma]$. Negation is only short-hand for an atom that matches all events in the input domain Σ except for events in the negated describable event set E. Note that this semantics differs from the more powerful negation operators found in other event algebras.

Concatenation. $\mathcal{C}_1\mathcal{C}_2$.



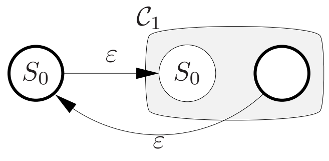
The concatenation operator detects a composite event matching expression $\mathcal{C}_1$ with a timestamp that *weakly* follows the timestamp of a composite event matching $\mathcal{C}_2$. The detection automaton for concatenation is constructed by connecting the generative state of $\mathcal{C}_1$ with a weak ϵ-transition to the initial state of $\mathcal{C}_2$.

Sequence. $\mathcal{C}_1; \mathcal{C}_2$.



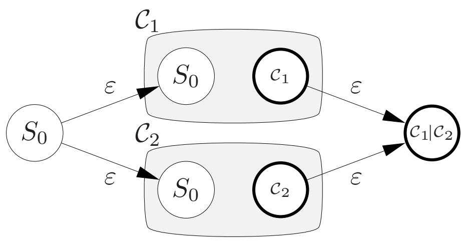
The sequence operator detects an event of type $\mathcal{C}_1$ *strongly* followed by an event of type $\mathcal{C}_2$. Unlike concatenation, this means that the interval timestamps of the events matching $\mathcal{C}_1$ and $\mathcal{C}_2$ must not overlap. The construction of the sequence detection automaton uses a strong transition for the ϵ-transition between the two sub-automata.

Iteration. $\mathcal{C}_1^*$.



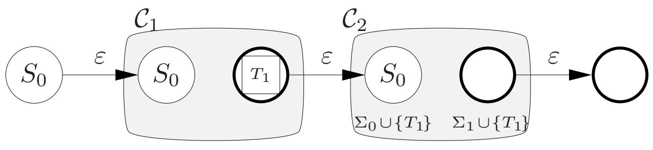
Any number of occurrences of $\mathcal{C}_1$ are matched by the iteration operator. Its detection automaton creates a loop from the generative state of $\mathcal{C}_1$ back to its initial state. If $\mathcal{C}_1$ receives an event that causes it to fail, then the composite expression $\mathcal{C}_1^*$ also fails.

Alternation. $\mathcal{C}_1 \,|\, \mathcal{C}_2$.



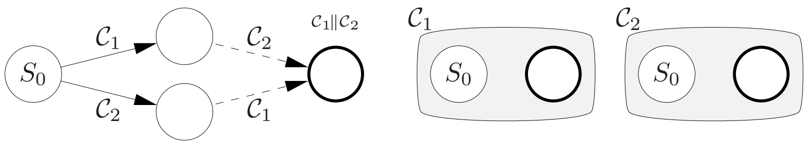
This composite event expression matches if either $\mathcal{C}_1$ or $\mathcal{C}_2$ is detected. The new automaton has an initial and a generative state with ϵ-transitions to both of the two sub-automata introducing nondeterministic behaviour.

Timing. $(\mathcal{C}_1, \mathcal{C}_2)_{T_1=tspec}$.



Timing relationships between composite events are supported by the timing operator that can detect event combinations within, or not within, a given time interval. This operator generates an event of type T_1 at the relative or absolute time specification *tspec* after a composite event of type $\mathcal{C}_1$ has been detected. The second expression C_2 may then use T_1 in its specification for atoms and input domains. Since time events are only locally visible, automata $\mathcal{C}_1$ and $\mathcal{C}_2$ must reside on the same node.

Parallelisation. $\mathcal{C}_1 \parallel \mathcal{C}_2$.



The final operator is parallelisation, which allows detection of two composite events $\mathcal{C}_1$ and $\mathcal{C}_2$ in parallel, only succeeding if both are detected. Unlike alternation, any interleaving of the two composite events is supported. The detection automaton for parallelisation is constructed by creating a new automaton that uses the composite events detected by $\mathcal{C}_1$ and $\mathcal{C}_2$ for its transitions.

7.5.1 Examples

The following are examples of valid expressions in the core composite event language. Let the describable event set A represent events corresponding to the subscription that 'Alice is in the office', let $\bar{A}$ be 'Alice has left the office', let B be 'Bob is in the office', and let P be 'anyone is in the office', as detected by an Active Bat.

1. $[A];[B]$. Alice enters the office followed by Bob.
2. $[A \subseteq \{A, B\}]$. Alice enters the office before Bob.
3. $([A], [B \subseteq \{B, T_1\}])_{T_1=1\,\mathrm{h}}$. Alice enters the office and Bob follows within 1 h.
4. $[\bar{A}]\,[\neg A \subseteq P]\,[A]$. Someone else enters the office when Alice is away.

7.5.2 Higher-Level Composite Event Languages

When designing a language for composite event detection, we have two conflicting requirements. On one hand, the language should be *machine-processable* so that it supports the efficient creation of composite event detection automata and the automatic decomposition of expressions for distributed detection. On the other hand, the syntax and semantics of the language should be high-level and intuitive, facilitating the task of writing expressions by programmers or end users. This means that the language should be *human-processable.* To unify these two requirements, we advocate the use of higher-level composite event languages for the specification of composite events in a natural and domain-specific way. Expressions from higher-level languages are then translated automatically into the core language described above. The following are three possible higher-level composite event languages.

Pretty Language. The "pretty" language has a more verbose syntax compared to the core language and resembles rule-based specification languages found in active database systems. It has a redundant set of operators and specifications are close to English language statements. A composite event specification, such as

```
Event_A followed_by Event_B within 1 hour,
```

makes it easier for non-programmers to use composite events.

Programming Language Binding. Similar to the XML-to-event binding of Hermes in Section 4.4.2, this binding provides programming language-specific access to composite event specification. It avoids having to deal with a special composite event language by allowing the construction of composite event expressions from method calls, such as

```
eventA.after(eventB.repeated(3)).
```

At runtime these method calls are translated into core composite event language expressions for detector construction.

Graphical Composition Model. In a ubiquitous computing environment, a user-friendly way for composite event specification is needed that makes it easy for users to interact with the system at runtime. Composite events, such as 'Turn the office light out after 7pm when the office is empty', can be described using a graphical composition tool that is based on a simple model familiar to users. For example, composite event streams could be visualised as water flows with different forms of piping for the construction of composite event expressions [HBWP01]. Alternatively, composite event detection could be modelled as the processing of events on conveyor belts in a factory [Nir03].

7.6 Distributed Detection

Our composite event service supports the distributed detection of composite events in an event-based middleware. This is achieved by decomposing composite event expressions in our core language into subexpressions that are detected by separate detectors distributed throughout the system. The support for the decomposition of composite event expressions allows popular subexpressions to be reused among event subscribers, thus saving computational effort and network bandwidth. In particular, the amount of communication is reduced because detectors for subexpressions can be positioned close to primitive event publishers that produce the events necessary for detection. Subexpressions can also be replicated for load-balancing and increased availability, and computationally expensive expressions can be decomposed to prevent any detector from becoming overloaded.

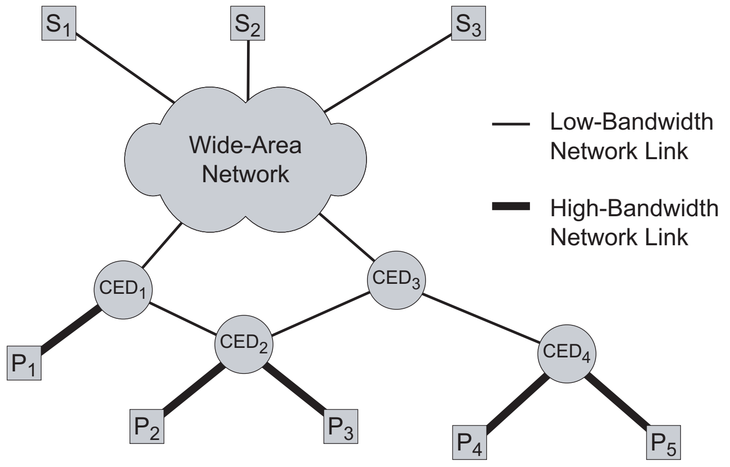


Figure 7.8: Illustration of distributed composite event detection

A system that benefits from distributed composite event detection is shown in Figure 7.8. The composite event detectors CED$_{1-4}$ for subexpressions are located close to the primitive event publishers P_{1-5} that publish events at a high rate and therefore must be connected through high-bandwidth network links. Low-bandwidth links in a wide-area network are used to connect the composite event subscribers S_{1-3}. The traffic on these network links is significantly lower because fewer event publications need to be transmitted after composite event detection. Since each detector subscribes to at most two event streams, no detector can get overwhelmed by the event rate.

The detection automata in our composite event service directly support distribution because they can subscribe to composite events detected by other automata in the event-based middleware. In Figure 7.9 the two automata $\mathcal{C}_1$ and $\mathcal{C}_2$ cooperate in order to detect the composite event expression $([B];[P]) \mid [M]$. The sub-automaton $\mathcal{C}_1$ detects the expression $[B];[P]$, which is then used by $\mathcal{C}_2$ in the event input domain and transition of state S_0. When this composite event is received, it causes a transition to the generative state S_1.

In the next section we present the capabilities of mobile composite event detectors. The behaviour of mobile detectors is controlled by the distribution policies in Section 7.6.2. Timing issues caused by the distributed detection of composite events are addressed by detection policies in Section 7.6.3.

7.6.1 Mobile Composite Event Detectors

A *mobile composite event detector* implements the distributed detection of composite events in our service. Mobile composite event detectors are agent-like entities co-hosted at event brokers

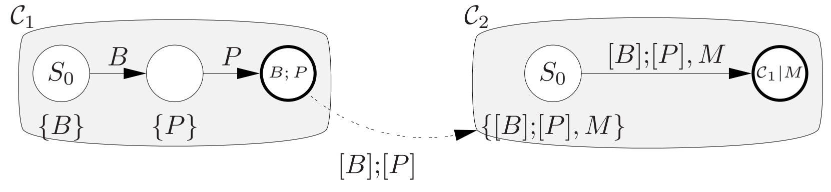


Figure 7.9: Two cooperating composite event detectors for distributed detection

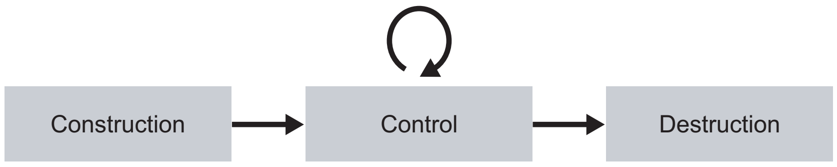


Figure 7.10: The life-cycle of a mobile composite event detector

that encapsulate one or more composite event detection automata for expressions from the core composite event language. They can subscribe to event publishers (and other mobile detectors) and publish the composite events detected by their automata. In addition, a mobile detector can move from one event broker to another in order to optimise the detection of composite events in the system.

When an event subscriber submits a new composite event subscription, a mobile detector is instantiated at an event broker and is then responsible for the detection of the new expression. The life-cycle of a mobile composite event detector is summarised in Figure 7.10. In the *construction phase*, the mobile detector establishes the detection of the new composite event subscription by cooperating with other existing mobile detectors. It then enters a *control phase*, during which the detection is optimised by adapting to dynamic changes in the environment and ensuring that it maintains compliance with distribution and detection policies described below. Finally, a *destruction phase* is reached when the mobile detector is no longer required because all event clients have unsubscribed or other detectors have made it redundant.

While in its control phase, a mobile detector can carry out several actions that are governed by distribution policies explained in the next section.

1. It can *instantiate* new automata for the detection of new composite event expressions or any subexpressions.
2. For distributed detection, it can decompose composite event expressions and *delegate* detection to other, already existing, mobile detectors.
3. The mobile detector can *migrate* to another event broker that, for example, is closer to the event publishers that the detector has subscribed to.
4. Finally, it can *destroy* any of its composite event detection automata that are no longer required.

7.6.2 Distribution Policies

A remaining difficulty is the decision on an optimal strategy for the decomposition of composite event expressions and the placement of composite event detectors in the system. This is complicated by the fact that the requirements for distributing detectors are potentially conflicting. For example, to minimise usage of network bandwidth, existing detectors should be reused for subexpressions as much as possible. However, if low notification latency is important, detectors should be replicated in various parts of the network, thus leading to increased bandwidth consumption. An optimal solution is a trade-off that takes the static and dynamic characteristics of the application and the network into account.

To make these trade-offs explicit, we introduce the notion of a *distribution policy*, which is a set of heuristics that governs the actions of mobile composite event detectors in the control

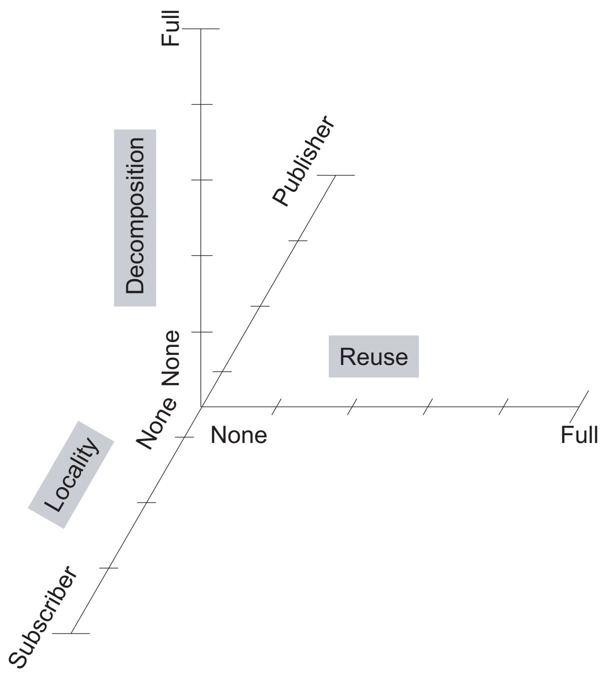


Figure 7.11: The design space for distribution policies

phase. Each composite event subscription submitted to the composite event service includes its own distribution policy for detection, depending on the application requirements of the event subscriber. During their lifetime, mobile composite event detectors attempt to comply with their distribution policy. Some distribution policies may require the aggregation of network or event broker statistics by mobile composite event detectors, such as communication latency or computational load. When defining distribution policies, three independent dimensions can be identified that help restrict the design space, as shown in Figure 7.11.

Decomposition. In order to reuse existing detectors, the degree of decomposition of a composite event expression must be stated in the distribution policy. Decomposition may increase the reliability of detection if multiple detectors are used for overlapping expressions. For load-balancing reasons, a complex expression may be decomposed into manageable subexpressions. The degree of decomposition ranges from none to full decomposition, where every possible subexpression is factored out. Some distribution policies only permit decomposition if there already exist detectors in the system that can be reused.

Reuse. The dimension of reuse specifies to what extent already existing detectors are reused for a new composite event expression or any of its subexpressions. Not reusing detectors leads to more reliability, whereas maximum reuse can save network bandwidth and computational effort. In situations in which detection latency is important, only detectors that are in close proximity should be reused.

Locality. The final dimension deals with the location of a new mobile composite event detector in the event-based middleware. The usage of network bandwidth can be reduced by moving detectors close to primitive event publishers, resulting in *publisher locality.* This avoids the wide dissemination of primitive events that may only be of interest to the composite event detector. The opposite approach, *subscriber locality*, is to put new composite event detectors close to composite event subscribers in order to improve detection latency.

Policy Name	Decomposition	Reuse	Locality
Minimum Latency	none	with locality only	subscriber
Minimum Bandwidth	for reuse only	maximum	publisher
Minimum Impact	for reuse only	maximum	none
Minimum Load	maximum	maximum	none
Maximum Reliability	for reuse only	at least n detectors	none

Table 7.1: Summary of five distribution policies

In practice, only certain combinations of these three dimensions result in useful distribution strategies. Five example policies are listed in Table 7.1 and described in the following.

Minimum Latency Policy. Detection latency is minimised by placing new detectors as close to primitive event subscribers as possible. Composite event expressions are not decomposed into subexpressions as this would increase detection latency. Existing detectors are only reused if they are local to event subscribers and detect the exact required composite events.

Minimum Bandwidth Policy. Bandwidth consumption of a composite event subscription is reduced by placing detectors close to primitive event publishers, relying on early filtering of event publications. Moreover, existing detectors are reused as much as possible so that no new network traffic is generated. Subexpressions are only created for reuse.

Minimum Impact Policy. The goal of this policy is to lower the impact of new detectors to the entire event-based middleware. Apart from minimising the bandwidth as before, computational load is spread out evenly among detectors. This means that new detectors do not have locality, and existing detectors are maximally reused.

Minimum Load Policy. Unlike the previous policy, a minimum load policy only minimises the load at any given composite event detector ignoring global impact and bandwidth usage. This is achieved by decomposing composite event expressions into the smallest possible subexpressions. The detectors are then evenly distributed across event brokers and already existing detectors are reused.

Maximum Reliability Policy. The final distribution policy attempts to improve the reliability of composite event detection by instantiating redundant detectors. At least n detectors have to exist for each subexpression and existing detectors may be reused. To avoid the clustering of detectors that might result in single points of failure, new detectors do not possess locality constraints.

7.6.3 Detection Policies

In a distributed system, events from different event publishers travel along separate network routes to mobile composite event detectors. Even if we assume that the network does not reorder event publications, out-of-order arrival of events at a detection automaton will occur because of the different associated network delays. This means that events in the event input sequence for a state are not correctly ordered with respect to the total order imposed by the interval timestamps. As a result, a *detection policy* is necessary that dictates when the next event in the event input stream can safely be consumed by the detection automaton. Note that premature consumption can lead to the incorrect detection or non-detection of composite events. Next we describe three possible detection policies for a composite event subscription.

Best-Effort Detection Policy. The best-effort detection policy states that events are consumed from the event input stream without delay and immediately cause a state transition or failure in the detection automaton. Although this policy may lead to incorrect detection, it is useful for subscribers that are more sensitive to detection delay than to bogus composite events.

Guaranteed Detection Policy. Under a guaranteed detection policy, an event is only consumed from the event input sequence once it has become *stable* [LCB99]. An event is considered stable if there is no other event with an earlier, totally-ordered interval timestamp in the system that should be part of this event input stream and be consumed next instead. When only stable events are used by a detection automaton, composite events cannot be missed or detected spuriously. A detection automaton knows that an event is stable when another event with a later timestamp from the same event publisher has been inserted in the event input stream. In case event publishers do not publish events at a high enough rate, dummy *heartbeat events* can be published that are used to “flush” the network and determine the stability of events.

Probabilistic Detection Policy. A disadvantage of the previous policy is that it introduces an unbounded delay at composite event detectors. For example, an event publisher may fail or decide not to cooperate by withholding heartbeat events. To avoid this problem, we propose a *probabilistic stability metric*, instead of a simple binary one. A detector attempts to estimate the probability that a given event in the event input stream is stable depending on the past history of event publications coming from that event publisher. An event is only consumed when its stability metric is above a certain threshold. This gives a good compromise between best-effort and guaranteed detection to composite event subscribers.

7.7 Implementation and Evaluation

In this section we describe the prototype implementation of the composite event service on top of JORAM [Obj02], an implementation of the Java Message Service (JMS) from Section 2.1.2. We have chosen JMS as a the basis for our implementation because JMS is a popular publish/-subscribe system and its lack of event-based middleware features demonstrates the wide applicability of our composite event service even to primitive publish/subscribe systems. JMS has the shortcoming that topics are centrally-hosted at a single JMS server without content-based routing of publication messages in an overlay broker network.

To support the automatic distribution of composite event expressions, all mobile composite event detectors subscribe to a common *administration topic* hosted by a well-known JMS server. For each expression, the detectors decide about the instantiation of new detection automata by exchanging messages through this topic. The locations of newly created detection automata are recorded in a JNDI [Sun99a] directory for further reference. Composite event types are entirely

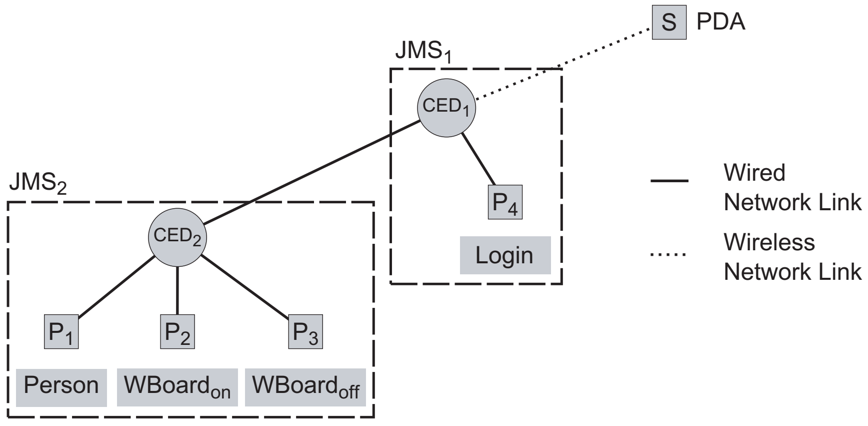


Figure 7.12: The network architecture of the Active Office experiment

hidden from JMS because composite events are published in topics whose names are constructed from the structure of the composite event types.

The efficiency of our composite event service in comparison with a naïve solution, in which composite event detection is left to event subscribers, is evaluated through an experiment in the Active Office application scenario. We assume that a user is interested in the list of participants and the electronic white-board content of meetings that she attends. This information should be sent to her PDA, that is connected via a low-bandwidth wireless network link, but only if she does not return to the workstation in her office within 5 minutes of the meeting. The goal of this scenario is to minimise the usage of the wireless link while maintaining a low notification delay.

The composite event subscription can be expressed in our core composite event language as

$$\begin{aligned}\mathcal{C} &\equiv ([\mathcal{C}_1(f_1)]\,,\ [T_1] \subseteq \{T_1, \texttt{Login}(f_2)\})_{T_1=5\,\mathrm{min}}\\ \mathcal{C}_1 &\equiv [\texttt{WBoard}_{\mathrm{on}}]\ \big[[\texttt{Person}(f_3)]\,[\texttt{Person}(f_3)]^*\,[\texttt{WBoard}_{\mathrm{off}}] \subseteq \{\texttt{Person}(f_3), \texttt{Board}_{\mathrm{off}}\}\big],\end{aligned}$$

where `Person`, $\texttt{WBoard}_{\mathrm{on}}$, $\texttt{WBoard}_{\mathrm{off}}$, `Login` are primitive events and $f_{1,2,3}$ are JMS filter expressions. Figure 7.12 shows how the composite event expression $\mathcal{C}$ is distributed by our composite event service over two composite event detectors CED$_1$ and CED$_2$ after factoring out the subexpression $\mathcal{C}_1$. Most primitive event topics and the topic for composite events of type $\mathcal{C}_1$ are hosted at the server JMS$_2$, and $\mathcal{C}$ and `Login` events are hosted at JMS$_1$.

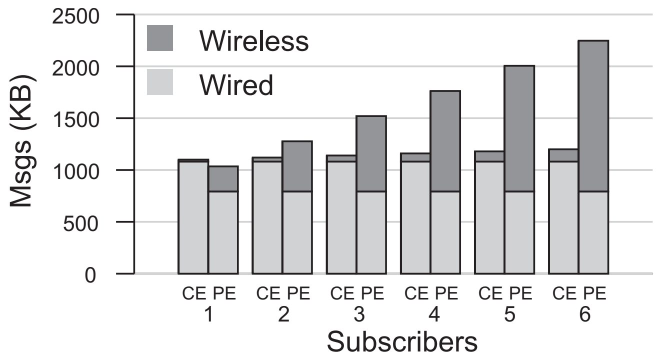


Figure 7.13: The amount of data sent in the Active Office experiment

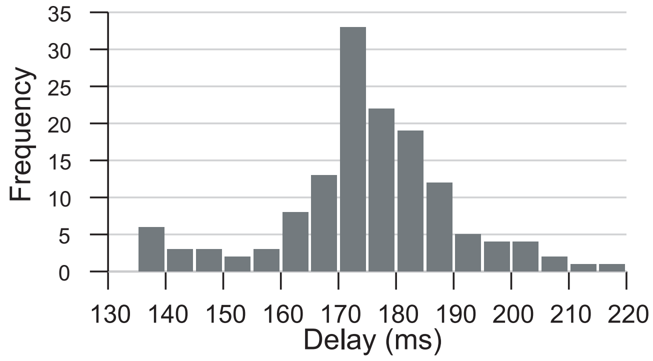


Figure 7.14: The delay distribution in the Active Office experiment

We compare our composite event service (CE) against a JMS-only solution (PE), in which the wireless PDA subscribes to all primitives events and performs the composite event detection in an ad-hoc manner. In Figure 7.13 the total amount of data transfered over the wireless and wired parts of the network is shown as a function of the number of composite event subscribers with expression $\mathcal{C}$. There is a small overhead of using distribution in our composite event service for a single subscriber. However, as the number of event subscribers increases, less data is sent over the wireless network because composite event detectors can be reused so that primitive events do not have to be sent over the wireless network to the PDA.

The additional notification delay introduced by our composite event service due to the involvement of multiple detectors remains small. The plot in Figure 7.14 shows the distribution of the delay for event subscribers in our experiment. The notification takes at most 220 ms after the composite event has logically occurred in the system and is dominated by the network latency.

7.8 Related Work

In this section we provide an overview of related work on composite event detection. Composite event detection first arose in the context of triggers in active database systems. Other related application areas are network systems monitoring and the interaction with ubiquitous computing environments. In general, distributed publish/subscribe systems leave the detection of composite events to the application programmer. An exception is SIENA (described in Section 2.2.2), that includes restricted *event patterns* without defining their precise semantics or giving a complete pattern language. A service for the detection of composite events using CORBA is presented by Liebig [LCB99]. Like our composite event service, it uses interval timestamps to make the uncertainty of timestamps in a distributed system explicit. The notion of event stability is introduced to handle communication delays.

Active Database Systems. Composite event detection in active database systems is usually not distributed. Early languages for triggers follow an event-condition-action (ECA) model [PD99, DBC96] and resemble database query algebras with an expressive, yet complex, syntax. In the *Ode* object database [GJS92], composite events are specified with a regular language and detected using finite state automata. Equivalence between the language and regular expressions is shown. Since a composite event has a single timestamp — that of the last primitive event that led to its detection, a total event order is established that does not deal with time issues. Composite event detectors based on Petri nets [Pet77] are used in the *SAMOS* database [GD94].

Coloured Petri nets can represent concurrent behaviour and store complex event data during execution. A disadvantage is that even for simple composite event expressions, Petri nets quickly become complicated. SAMOS does not support distributed detection and has a simple time model. The motivation for *Snoop* [CM93] was to design an expressive composite event language with temporal support. A detector in Snoop is a tree that mirrors the structure of the composite event expression. Its nodes implement language operators and conform to a given *consumption policy*. A consumption policy determines the semantics of operators by resolving the order in which events are consumed from an event history. For example, under a *recent* consumption policy only the event that most recently occurred is considered and others are ignored. Detection then propagates up the tree with the leaves being primitive event detectors. A drawback of this approach is that detectors are Turing-complete, which makes it difficult to estimate their resource usage in advance. In addition, consumption policies influence the semantics of operators in a non-intuitive and operator-dependent way. For simplicity we have decided to only support a *chronicle* consumption policy.

Distributed Systems Monitoring. Similar to network systems monitoring in Section 7.2, composite events can be used for the monitoring of distributed systems. Schwiderski presents a distributed composite event monitoring architecture [Sch96] based on the 2g-precedence time model. This model makes strong assumptions about the clock granularity that are not valid in large-scale, loosely-coupled distributed systems. The composite event language and detectors are similar to Snoop and suffer from the same shortcomings. The work addresses the issue of delayed events in distributed detection by *evaluation policies*. *Asynchronous* evaluation allows a detector to consume an event without delay, whereas *synchronous* evaluation forces it to wait until all earlier events have arrived, as indicated by a heartbeat infrastructure. Although the detection can be made distributed, the placement of detectors in the system is left to the user. The *GEM* system [MSS97] has a rule-based event monitoring language. It also follows a tree-based approach and assumes a total time order. Communication latency is handled by annotating rules with tolerable delays, which is not feasible in an environment with unpredictable delays, such as a large-scale distributed system.

Ubiquitous Systems. Research efforts in ubiquitous computing have resulted in composite event languages that are intuitive to use by users of environments such as the Active Office. The work by Hayton [Hay96] on composite events in the Cambridge Event Architecture defines a language that is targeted at non-programmers. Push-down finite state automata are used to detect composite events, but the semantics of some of the operators is non-intuitive. Although detection automata can use composite events for input, distributed detection is not handled explicitly and only scalar timestamps are used in the time model.

7.9 Summary

In this chapter we described a distributed higher-level service for composite event detection in an event-based middleware. We explained our idea of composite events and motivated their use with examples from three application scenarios. We then introduced the layered design of our service, each layer providing different levels of expressiveness. Composite events are detected by automata that support distribution and a flexible time model for composite events. Event subscribers of the composite event service use a core composite event language to specify event patterns using a series of operators. We also gave examples for three higher-level specification languages for composite events that are domain-specific. After that, we described distributed composite event detection by presenting the features of our mobile composite event detectors and explained how they are controlled by distribution and detection polices. The design space for distribution polices gives rise to a variety of different strategies for distributing composite event expressions. Finally, the efficiency of our service was demonstrated with an experiment from the Active Office scenario.

8 Security

The third service presented in this thesis adds security to an event-based middleware. Security has received little attention in publish/subscribe systems so far and, unlike the previous services, it affects many different parts of an event-based middleware. We propose a security service [BEP^{+}03] that uses role-based access control to provide three mechanisms: restrictions on the interaction of event clients with the middleware, trust levels for event brokers, and the encryption of event data to control information flow in the middleware on a fine-grained basis. An advantage of our approach is that we do not require separation of the overlay broker network into distinct trust domains. The prototype implementation of our security service is built on top of Hermes.

The security service is influenced by the security needs of the two applications scenarios discussed in the next section. In Section 8.2 we define the requirements of a security service, showing how publish/subscribe communication impacts on security. The secure publish/subscribe model implemented by our service is introduced in Section 8.4. It includes boundary access control using restrictions, different-levels of event broker trust, and encryption of event attributes. The chapter finishes with an overview of a prototype implementation (Section 8.5) using Hermes, a brief evaluation (Section 8.6), and related work on security in publish/subscribe systems in Section 8.7.

8.1 Application Scenarios

In this section we revisit the two application scenarios from Section 1.2 and examine how they motivate the need for security in an event-based middleware. When considering security, we focus on issues of access control to the middleware and confidentiality of the event data being disseminated in the system.

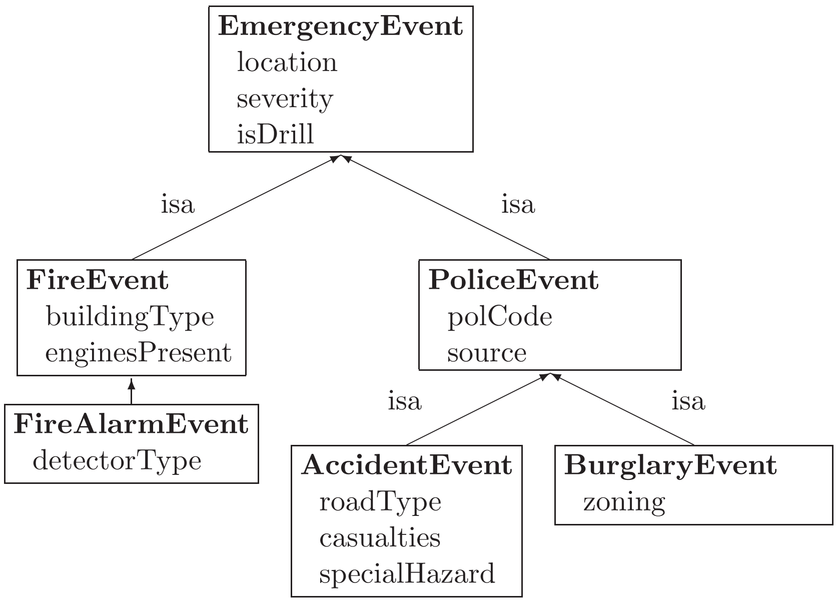


Figure 8.1: An event type hierarchy for the Active City

8.1.1 The Active City

The *Active City* is an extension of our previous Active Office environment to a geographically larger system covering an entire city. In an Active City, different city services, such as police and fire departments, ambulances, hospitals, and news agencies, cooperate using a shared event-based middleware for information dissemination. Since these city services are under separate management and each have individual security implications, the event-based middleware must be flexible enough to accommodate a wide range of security policies and mechanisms to enforce them.

An excerpt of an event type hierarchy with event attributes that could be employed by cooperating services in an Active City is shown in Figure 8.1. Information about a road traffic accident reported to the police in an `AccidentEvent` should be visible to the emergency services so that an ambulance can be dispatched if there are any casualties, but only anonymised data should be passed on to a news agency. The challenge is that some information may flow freely through the Active City, whereas other information has to be closely controlled. A naïve solution would be for each city service to operate a separate, trusted event-based middleware deployment with controlled gateways between networks, forming an event federation [Hom02]. However, this would result in complex policy management at the gateways, a significant waste of resources due to redundancy, and an increased event notification delay between services. It would also prevent event clients from one domain using the infrastructure of another while roaming.

8.1.2 News Story Dissemination

In an Internet-wide system for the dissemination of news stories, it is important that customers only receive the service that they are paying for. For example, a customer who has subscribed to a premium service should receive up-to-date news bulletins without delay, as opposed to a standard service subscriber that can only see events relating to older news reports. Moreover,

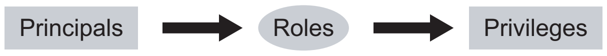


Figure 8.2: The role-based access control model

subscribers should only be allowed to subscribe to the news topics that they are entitled to. To ensure this, it is not sufficient to merely rely on subscriptions in the event-based middleware because event brokers that perform content-based routing of news events may be under the administration of customers and thus not trusted to honour subscriptions correctly. Using partially trusted event brokers for event dissemination in customer networks is otherwise in the interest of news agencies because it reduces the resource requirements of their middleware deployments. When the service subscription of a customer changes, the event-based middleware should quickly adapt to the change in policy.

8.2 Requirements for Security

Security mechanisms for an event-based middleware differ from traditional middleware security because of publish/subscribe communication semantics. Many-to-many interaction in an event-based middleware mandates a scalable access control mechanism. The anonymity of the loose coupling between event publishers and subscribers makes it difficult to use standard security techniques, such as access control lists, since principals can often not be identified beforehand. Content-based routing of events conflicts with the encryption of data because an event broker must have access to the content of an event for its routing decision [WCEW02]. Any access control mechanism should incur little overhead at publication time because event publications may have a high rate and thus routing should be carried out as quickly as possible.

Since event clients are not trusted, a security service should include perimeter security to control access of event clients to the event-based middleware. As seen in the application scenarios, event brokers are trusted to cooperate for the sake of event dissemination but they may not be allowed to see all event data. Different levels of event broker trust are necessary and must come with mechanisms to remove compromised event brokers. The confidentiality of data stored in event attributes must be preserved even in the light of event matching and content-based routing. At the same time, as much as possible of the overlay broker network should be used for event dissemination so that a single infrastructure for both public and private information exists in order to improve efficiency, administrability, and redundancy in the event-based middleware.

8.3 Role-Based Access Control

We decided to follow the *role-based access control* model (RBAC) [SCFY96] in our security service. RBAC simplifies security administration by introducing *roles* as an abstraction between *principals* and *privileges*, as shown in Figure 8.2. Roles permit principals and privileges to be grouped intuitively in the system and addresses the anonymity of event clients in an event-based middleware. This grouping increases scalability of the access control mechanism because there are fewer roles than principals and privileges in the system. To obtain privileges, a principal such as an event publisher or subscriber presents credentials that allow it to acquire a role membership that is associated with the desired privileges.

The *Open Architecture for Secure Interworking Services* (OASIS) [BMY02] is a distributed implementation of a role-based access control model that is used in the security service for access control decisions. It includes an expressive policy language to specify rules for role acquisition. Due to its session-based approach, event communication is used to revoke currently active roles of principals in a timely manner when prerequisite credentials are revoked. The OASIS implementation uses X.509 certificates [ITU00] for authentication and proof of role membership.

8.4 The Secure Publish/Subscribe Model

In this section we describe our secure publish/subscribe model for an event-based middleware. As a general design philosophy, the model couples access control with event types. Since the event space is already structured into event types, it is intuitive to leverage this for the specification of access control policy, but more fine-grained specification in terms of event attributes and type- and attribute-based subscriptions is also supported by the model.

An example of an event-based middleware deployment that implements the secure publish/-subscribe model is given in Figure 8.3. There are three mechanisms in the model to accomplish access control. First, boundary access control to the middleware, as described in the next section, is achieved by controlling access of event clients to local event brokers with an OASIS policy. Middleware services requested by event clients can either be granted, rejected, or partially granted after imposing restrictions. In Section 8.4.2, the second mechanism assigns an event broker to a particular trust category that prescribes the types from the event type hierarchy that the event broker is permitted to handle. Finally, confidential event attributes in an event publication are encrypted, limiting access to those attributes so that a single event publication can contain both public and private data. Content-based routing decisions on encrypted event attributes can only be carried out by event brokers that possess the necessary decryption key. Event attribute encryption will be explained in Section 8.4.3.

8.4.1 Boundary Access Control

Local event brokers that host event clients are OASIS-aware and perform access control checks for every request made to them. This ensures that only authorised clients have access to the event-based middleware in compliance with access control policies. As shown in Figure 8.3, local event brokers delegate the verification of credentials passed to them by event clients to an OASIS engine. Four types of OASIS policy are employed to restrict the actions of event clients. The

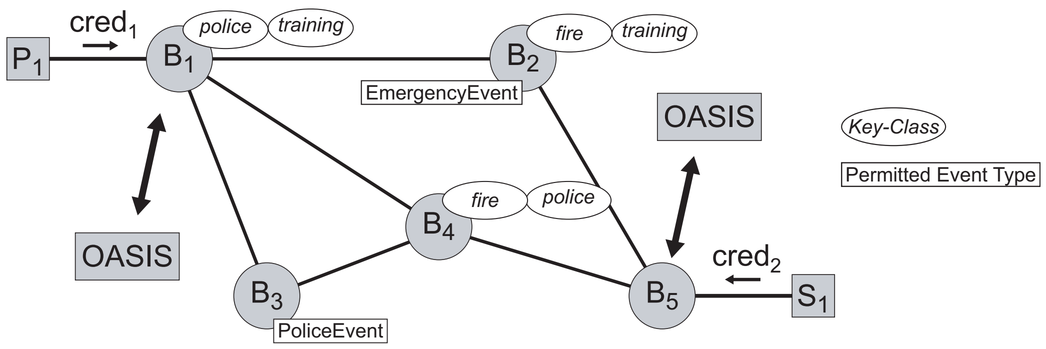


Figure 8.3: Illustration of the secure publish/subscribe model

event client that creates a new event type becomes its *event type owner* and is then responsible for specifying policy.

Connection Policy. This policy states the required credentials for an event client to be permitted hosting by a given event broker. A client can only invoke event-based middleware services if it maintains a connection with at least one local event broker.

Type Management Policy. The creation, modification, and removal of event types in the event type hierarchy is controlled by a type management policy. Usually, credentials certifying that an event client has the role of event type owner for an event type allow it to perform type management. This also avoids conflicts between clients from different applications.

Advertisement Policy. For every event type in the system, an advertisement policy specifies the roles an event publisher must acquire in order to advertise events of this type. This policy is generally specified by the event type owner.

Subscription Policy. Similarly, a subscription policy lists the necessary roles for an event subscriber to subscribe to events of that type. The policy may also prescribe the content-based filter expressions that are permitted and is again defined by the event type owner.

When an event client violates the connection or type management policies, the event broker rejects the operation invoked by the client. For advertisement and subscription policies, certain requests may be partially accepted by imposing a *restriction* on the original event advertisement or subscription. An advertisement restriction limits the advertisement by restricting the events that the event publisher is allowed to publish. Likewise, a subscription restriction transforms the client-requested subscription into a different, less powerful one. The client may or may not be notified by its local event broker that a restriction has been imposed for privacy reasons. The secure publish/subscribe model supports two flavours of restrictions.

Publish/Subscribe Restrictions. This kind of restriction takes the original submitted advertisement or subscription and replaces it by a different, more limited one. In the case of an event advertisement, the event type in the advertisement is replaced by a less specific parent type from the event type hierarchy. For event subscriptions, the publish/subscribe restriction specifies an upper bound on the event type and content-based filtering expression that the event subscriber is allowed to submit. If the submitted subscription is covered by the subscription restriction, the subscription is accepted without change, otherwise it is automatically downgraded to the restricted subscription.

Generic Restrictions. A generic restriction is not expressible by the event-based middleware since it can include any predicate evaluations permitted by OASIS. Although the original advertisement or subscription submitted by the event client is passed on to the event-based middleware, all later events are restricted according to the arbitrary predicate function in the generic restriction. For example, a generic advertisement restriction may reject the publication of events with certain content, and a generic subscription restriction may perform additional filtering of events on the message size of the event publication, which otherwise could not be expressed in an event subscription.

The advantage of publish/subscribe restrictions is that they do not incur an overhead during the dissemination of events. Since the original advertisement or subscription is replaced by a more limited version, the event-based middleware implicitly enforces policy and no events need to be dropped at client-hosting brokers. The same is not true for generic restrictions because their additional expressiveness comes with the price of having to evaluate arbitrary predicates at client-hosting brokers to decide whether an event client can publish or be notified of a given event publication.

8.4.2 Event Broker Trust

The previous mechanism for boundary access control using restrictions assumes that all event brokers are equally trusted to process data, which is not true in practice. When an event broker joins the event-based middleware, it authenticates with its credentials and is then believed to participate correctly in the routing of events according to the type- and attribute-based routing algorithm. It maintains encrypted network connections with its neighbouring event brokers in the overlay broker network. However, an event broker may not be trusted enough to gain access to data in particular event publications or subscriptions. To make these trust relationships explicit, event brokers are associated with event types from the event type hierarchy that they are permitted to handle. This is illustrated in Figure 8.3, where event broker B_3 is only permitted to process events of type `PoliceEvent`. Event brokers may be authorised to handle all event types that are more specific or more general than a given type, in other words are sub- or supertypes of an event type.

When routing event advertisements, subscriptions, and publications in the overlay broker network, an event broker only passes on a message to the next event broker after obtaining proof in the form of a role membership certificate that this event broker is authorised to handle that particular event type. Otherwise, the event broker is forced to make a different routing decision. This can be done by acting as though the untrusted event broker has failed, relying on the fault-tolerance properties of event routing in Hermes that will ensure a different routing path. Note that event broker trust encompasses the handling of entire event types only, but we relax this restriction through the use of event attribute encryption, as described in the next section.

8.4.3 Event Attribute Encryption

Event broker trust from the previous section excludes brokers that are not trusted to handle specific event types from routing. As argued before, this coarse-grained approach effectively splits the overlay broker network into several trust domains, thus weakening the reliability and efficiency of event routing. A better solution is to prevent an untrusted event broker from accessing confidential data but still enabling it to perform content-based routing on other attributes. We achieve the goal of a single event that can hold private and public information by encrypting event attributes in event publications with different cryptographic keys. Although this introduces a larger runtime overhead due to cryptographic operations during event routing, this is justifiable as it leads to more expressive access control specifications where event types no longer have to be strictly divided into private and public categories. Another advantage of this scheme is that access control policy can also associate event clients, which have access privileges, to event attributes.

In addition to event types, event brokers are also trusted with a number of *key-classes*. A key-class is a collection of cryptographic keys for encrypting event attributes, that supports key rotation and revocation. Access control to individual event attributes is achieved by signing and encrypting them with a key from a given key-class so that only trusted event brokers can decrypt these attributes. An event broker can only read or write an event attribute if it has a role

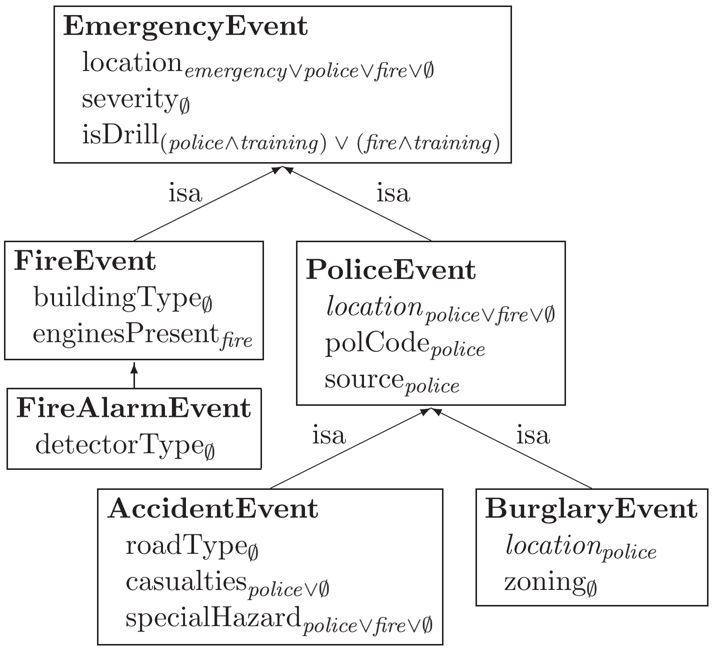

EmergencyEvent
location$_{emergency \vee police \vee fire \vee \emptyset}$
severity$_{\emptyset}$
isDrill$_{(police \wedge training) \vee (fire \wedge training)}$
isa
isa
FireEvent
buildingType$_{\emptyset}$
enginesPresent$_{fire}$
FireAlarmEvent
detectorType$_{\emptyset}$
PoliceEvent
location$_{police \vee fire \vee \emptyset}$
polCode$_{police}$
source$_{police}$
isa
isa
AccidentEvent
roadType$_{\emptyset}$
casualties$_{police \vee \emptyset}$
specialHazard$_{police \vee fire \vee \emptyset}$
BurglaryEvent
location$_{police}$
zoning$_{\emptyset}$

Figure 8.4: An event type hierarchy with attribute encryption

membership that includes access to the appropriate key-classes. This also means that an event client can only submit an event subscription or publication that refers to encrypted attributes to its local event broker if it can prove that it owns credentials for the required key-classes. The event broker then performs the cryptographic operations on the client's behalf.

To include event attribute encryption in the Hermes event model, the event type hierarchy is extended with a description of the key-classes that are necessary to access the content of event attributes, as shown in Figure 8.4. Each event attribute is annotated with its key-classes in disjunctive normal form. A conjunction of key-classes means that the attribute is encrypted with keys from several key-classes in sequence. For example, the `isDrill` attribute in the `EmergencyEvent` type has to be either encrypted under the *police* and *training*, or under the *fire* and *training* key-classes. This prevents anyone receiving emergency-related events in the Active City from finding out whether this is an exercise drill unless they are a training instructor with access to the *training* key-class. Unencrypted event attributes are denoted with the empty key-class $\emptyset$. In Figure 8.3 event brokers are annotated with the key-classes that they are permitted to use.

Note that the standard subtyping relation between event types must still hold so that a subtype is more specific than its parent type. As a result, key-classes can only be removed from inherited event attributes but never added. This is illustrated with the `location` attribute whose access becomes more restrictive as new event types are derived.

Encrypted Attribute Coverage

When an event subscriber submits a type- and attribute-based subscription for an event type with encrypted attributes, attribute predicates in the subscription must also be encrypted with appropriate key-classes for the subscription to match events. The subscriber selects one or more key-classes, from all the key-classes for which it is authorised to use, for the encryption of the

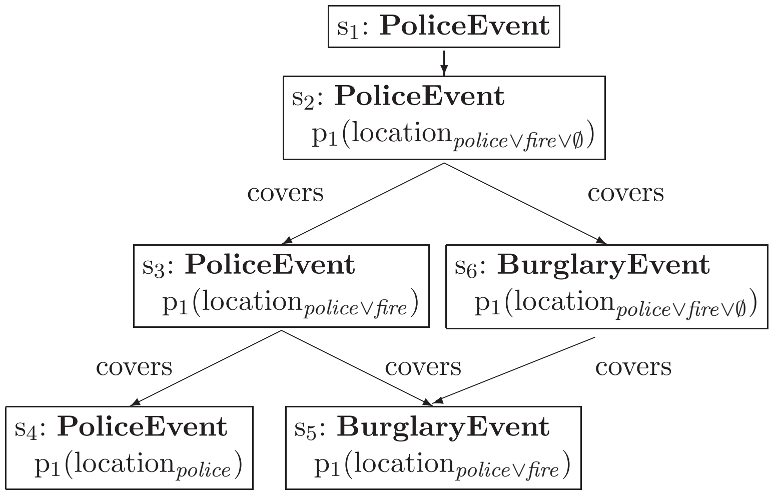

s1: PoliceEvent
s2: PoliceEvent p1(location police∨fire∨∅)
covers
covers
s3: PoliceEvent p1(location police∨fire)
s6: BurglaryEvent p1(location police∨fire∨∅)
covers
covers
covers
s4: PoliceEvent p1(location police)
s5: BurglaryEvent p1(location police∨fire)

Figure 8.5: Subscription coverage with attribute encryption

attribute predicate. As a consequence, the event model from Section 3.3.1 must be extended to support a coverage relation between event subscriptions and publications, and among event subscriptions that use attribute encryption. Informally, an encrypted attribute predicate can only be matched by an encrypted event attribute in a publication if it was encrypted with the same key-classes. When an attribute predicate should match attributes encrypted under several different key-classes, it must be disjunctively encrypted multiple times using these key-classes and several copies of the attribute predicate must be included in the subscription. For coverage among subscriptions, an attribute predicate encrypted under particular key-classes is covered by another encrypted predicate if the second predicate covers the first and is encrypted under at least the key-classes of the first predicate.

Definition 8.1 (Encrypted Attribute Coverage) *An encrypted event attribute a^K is* covered *by (or* matches*) an encrypted attribute predicate p^L,*

$$a^K \sqsubseteq p^L,$$

if and only if

$$a \sqsubseteq p \wedge K \subseteq L$$

holds, where K is the set of key-classes under which a is conjunctively encrypted and L is the set of a conjunction of key-classes under which p is disjunctively encrypted. An encrypted attribute predicate $p_1^{L_1}$ is covered *by another encrypted attribute predicate $p_2^{L_2}$,*

$$p_1^{L_1} \sqsubseteq p_2^{L_2},$$

if and only if

$$\forall a.\ a \sqsubseteq p_1 \Rightarrow a \sqsubseteq p_2 \wedge L_1 \subseteq L_2,$$

holds, where a is an event attribute and L_1 and L_2 are sets of conjunctions of key-classes with disjunctive encryption.

We illustrate this updated coverage relation in Figure 8.5, showing six example subscriptions with regard to the previous event type hierarchy. Subscription s_1 is most generic because it does not include any attribute predicates. The attribute predicate in subscription s_3 does not match events with an unencrypted `location` attribute and therefore s_3 is covered by s_2. Subscription s_4 is most specific because the attribute predicate is only encrypted under the *police* key-class.

8.5 Implementation

As a proof of concept, the security service was implemented as part of the Hermes implementation in DSSim and interfaced with an OASIS RBAC implementation. In this section we present how symmetric and asymmetric cryptography is used to implement the secure publish/subscribe model, and we have a closer look at the interaction of the security service with Hermes.

8.5.1 Cryptographic Techniques

The implementation of the secure publish/subscribe model uses several different types of keys (for symmetric cryptography) and key pairs (for asymmetric cryptography) stored at event brokers and clients. Asymmetric key pairs are used for initial authentication of event brokers and clients using the OASIS implementation.

Event Client Asymmetric Key Pair k_{EC}. Event clients require an authentication key pair in the form of an X.509 identity certificate for their interaction with a local event broker. This enables event clients to acquire OASIS role memberships so that local event brokers can verify policy governing boundary access control and access to key-classes.

Event Broker Asymmetric Key Pair k_{EB}. Likewise, event brokers have an authentication key pair to acquire OASIS role memberships that allow them to join the overlay broker network, maintain encrypted SSL connections with their neighbouring event brokers, and obtain access to event types and key-classes.

Event attribute encryption uses key-classes that contain symmetric keys for the fast decryption of event attributes during event routing. A key-class is managed by an event broker that acts as the *key-master* for the key-class. Since event clients in Hermes are less trusted than event brokers, symmetric keys of a key-class are only kept at event brokers that may encrypt or decrypt data on behalf of authorised event clients. Key rotation within a key-class is regularly initiated by key-masters and provides new symmetric encryption keys to authorised event brokers so that compromised event brokers or keys can be excluded.

Key-Class Symmetric Key $\bar{k}_{\mathrm{KC}}$. A current symmetric key of a key-class is used to encrypt and decrypt event attributes. An event attribute is signed and encrypted at the local event broker hosting the event publishers that published the event. It may then be decrypted at every authorised event broker along its routing path that requires access to the event attribute in order to evaluate attribute predicates. Before delivery to an authorised event subscriber, the local event broker decrypts the event attribute on behalf of the client.

Key-Master Asymmetric Key Pair k_{KM}. The key-master for a given key-class has an authentication key pair so that it can authorise the rotation of symmetric keys of the key-class. Updated symmetric keys are signed by the key-master prior to distribution to event brokers that require access to the key-class.

For reliability reasons, key-masters are replicated and follow an election protocol to choose a leader that initiates key rotations. A new symmetric key $\bar{k}_{\mathrm{KC}}$ for a key-class is distributed to all authorised event brokers by first signing it with the key-master key k_{KM} and then encrypting it individually with the identity keys k_{EB} of all event brokers that should have access to the key-class. A standard event publication disseminates the package of encrypted copies of the updated symmetric key $\bar{k}_{\mathrm{KC}}$ to event brokers, which act as event subscribers for key-class update events, which are another type of meta-event.

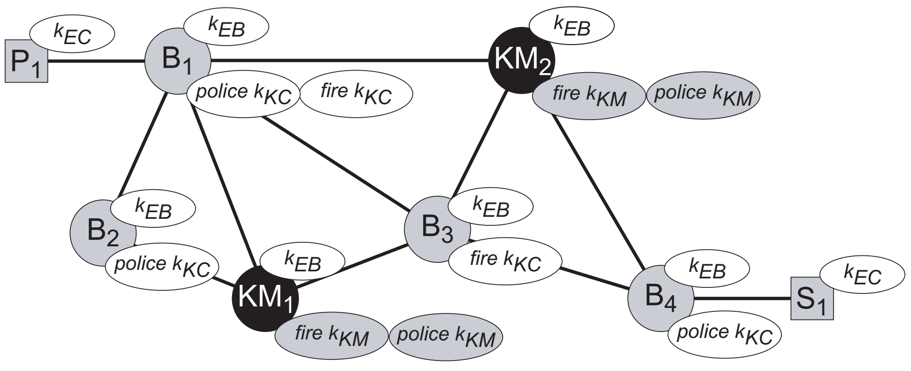


Figure 8.6: Cryptographic keys and key-masters for key-classes

An example of a system with two key-classes *police* and *fire* and two replicated key-masters KM_1 and KM_2 managing both key-classes is shown in Figure 8.6. Event brokers and clients are annotated with their cryptographic keys, as described above.

8.5.2 Hermes Integration

The implementation of the security service closely interacts with HERMES and takes advantage of peer-to-peer routing with rendezvous nodes. The rendezvous node for an event type must be trusted with the event type that it manges. It also functions as a policy repository for boundary access control that contains the type, advertisement, and subscription polices, which are signed by the event type owner's authentication key. For faster OASIS policy verification, client-hosting brokers may keep a local policy cache that is updated using *policy evolution events*. These meta-events are disseminated from the event type owner to all interested client-hosting event brokers to update their local policy cache during policy evolution.

Key-masters in the system are installed with help of the distributed hash table provided by PAN. The key-master for a key-class resides at the event broker whose nodeID is numerically closest to the hash of the key-class name. Similar to rendezvous node replication, event brokers from the leaf set of the primary key-master are chosen to host replicas. Meta-events are used by key-masters to distribute updated symmetric encryption keys for a key-class to other event brokers.

Our secure publish/subscribe model states that only event attributes can be encrypted but not event type names. This could potentially result in undesired leakage of information to event brokers and clients. However, this is not the case in the HERMES event routing algorithm, because publication and subscription messages are routed according to the SHA-1 hash of the event type name, thus protecting the clear-text event type name from unauthorised access.

8.6 Evaluation

Referring back to the two applications scenarios that motivated security in an event-based middleware, our secure publish/subscribe model provides sufficient mechanisms to implement both applications. We set up an experiment in the context of an Active City that compared selective attribute encryption, as supported by our service, against a naïve approach, where the entire event publication is encrypted several times with all relevant keys to achieve the same behaviour. In such an approach, the same event has to be published multiple times, once for every security domain in the event-based middleware [PPF+03]. As expected, attribute encryption results in a substantial saving of network bandwidth because fewer event publications need to be disseminated by the event-based middleware.

8.7 Related Work

In this section we provide a brief overview of previous work in the area of security in publish/-subscribe systems. Preliminary work on security issues under publish/subscribe semantics can be found in [WCEW02]. It identifies the necessity for ensuring the confidentiality of event publications and subscriptions and suggests accountability for billing purposes, however, no mechanisms are provided. In the work by Miklós [Mik02], upper bound filters on advertisements and subscriptions in Siena are proposed but the confidentiality of event publications within the publish/subscribe system is not guaranteed. The Narada Brokering project includes a distributed security framework [PPF+03, YHF+03] that uses access control lists to control event publishers and subscribers for a topic, limiting the scalability. Cryptographic keys for encrypting publications are centrally managed by a Key Management Centre (KMC). An event publisher can choose to use a central topic key from the KMC or the public keys of all event subscribers for encryption, which contradicts decoupled publish/subscribe semantics. Access control can only be provided at the granularity of whole events, and event brokers are implicitly trusted, rather than using different trust levels as supported by our security service.

8.8 Summary

The final higher-level service for an event-based middleware described in this thesis deals with security. We motivated the need for security in an event-based middleware with two application scenarios that require fine-grained access control to events, even with a partially-trusted infrastructure of event brokers. The main part of the chapter presented our secure publish/subscribe model that leverages role-based access control to regulate access to the event-based middleware. Within the middleware, event brokers have different levels of trust that map to the event types that they can process. Event attributes can be encrypted using key-classes and the coverage relation in the event model was extended to reflect this. We finished the chapter with a description of our prototype implementation of the secure publish/subscribe model and summarised how cryptographic keys were used and how the service interacted with Hermes.

9 Conclusions

With distributed systems becoming increasingly large-scale and dynamic, the nature of applications has changed substantially over the last decade. Traditional distributed systems arose from research into LAN-based systems and were based on assumptions such as long-lived group membership, mostly reliable communication, and one-to-one interaction between entities. These no longer hold for today's large-scale, Internet-wide, heterogeneous systems. The recent proliferation of peer-to-peer applications is a clear indication of this, and their need to handle highly dynamic participants and communication faults has led to novel routing algorithms and fault-tolerance mechanisms. These approaches can be used to make any large-scale distributed system more robust and adaptive to dynamic environments, thus embracing principles from autonomic computing [KC03].

In this thesis we addressed an important issue when developing large-scale distributed systems for deployment on the Internet: that of having proper middleware support, which handles the communication needs of application clients in a scalable and efficient way, and all without compromising traditional middleware features. We argued that event-based middleware can provide this support and described Hermes, a distributed, event-based middleware that employs peer-to-peer techniques for scalable and robust event dissemination. Hermes leverages peer-to-peer techniques for managing its overlay network of event brokers and adding fault-tolerance to its event dissemination algorithms. We found that a distributed hash table is a powerful data structure for application-level routing algorithms in large-scale distributed systems.

A core middleware platform that provides only communication support is not sufficient because the requirements of large-scale distributed applications differ. It is therefore essential that a middleware platform, such as Hermes, is extensible through higher-level middleware services in a distributed computing environment. To demonstrate aspects of a fully-featured event-based middleware, we introduced three middleware services that give evidence of the flexibility and extensibility of this type of middleware. A congestion control service avoids congestion in the overlay broker network and thus enables a resource efficient deployment of the middleware. The expressiveness of subscriptions is enhanced with a composite event service that detects complex patterns of events, allowing event clients to specify their interests more precisely and to control high volume event dissemination. Finally, a security service, which supports events carrying both public and private data, integrates fine-grained access control with the event-based middleware, addressing the access control and privacy needs of applications. These services facilitate large-scale application development because an application programmer can rely on their help when having to cope with the complexity of a heterogeneous and dynamic environment.

9.1 Summary

We began with an overview of the requirements of an event-based middleware if it is to support a variety of dynamic, large-scale applications. We then investigated the functions and features of this novel type of middleware, which led to five design models that helped structure the design space in terms of event data, components, routing, reliability, and services. Particular emphasis was placed on the programming language integration of the middleware, which resulted in the inclusion of event types and type inheritance into the event model. Application-level event dissemination in the routing model was extended with fault-tolerance mechanisms from the reliability model.

Our primary goal was the development of Hermes, our event-based middleware platform. We described its layered architecture and the two event routing algorithm supported by Hermes, type-based routing, which supports subscriptions according to an event type, and type- and attribute-based routing, which provides content-based filtering on event attributes as well. Both routing algorithms use rendezvous nodes to construct scalable event dissemination trees on top of a distributed hash table. Because of our requirement of programming language integration, we extended the routing algorithms with event type inheritance and support for supertype subscriptions. We also described the fault-tolerance mechanisms in the algorithms that are based on a soft-state approach and the replication of rendezvous nodes. A prototype implementation of Hermes was presented in more detail, as was an evaluation of Hermes routing in a distributed systems simulator, comparing it with the Siena routing algorithm, which is standard for content-based routing of events. The experiments showed that Hermes routing builds efficient event dissemination trees and minimises the state stored in routing tables at event brokers.

The issue of congestion in an event-based middleware was addressed by our congestion control service. This service takes into account the special properties of congestion control in overlay broker networks. It is able to perform congestion control during normal operation and recovery after failure. The congestion control mechanism consists of a publisher-driven, and a subscriber-driven algorithm that regulate the rate of event publications and NACK messages in order to prevent congestion from occurring. We also described the implementation of the two algorithms as part of the Gryphon event broker and evaluated their behaviour with realistic experiments. The evaluation proved that the algorithms can quickly adapt to congestion in the system and change system parameters to compensate for whatever resource shortage is causing congestion.

We argued that ubiquitous computing applications, such as the Active Office, benefit from more expressive subscriptions in the form of composite event subscriptions. Our second service for an event-based middleware provides the distributed detection of complex event patterns. We introduced the two main parts of the service: composite event detection automata, that detect patterns with limited resource requirements, and a core composite event language, that allows the specification of event patterns with operators similar to regular expressions. The composite event service improves on previous work on pattern detection in distributed systems by supporting the decomposition and distribution of complex composite event expressions. Distribution and detection polices control the behaviour of distributed detectors according to the requirements of event subscribers. An experiment in the Active Office scenario showed that our generic composite event service operates with negligible overhead compared with a client-side implementation. This work indicates a way to approach the problem of managing environments with potentially excessive volumes of primitive events.

The final service developed in this thesis addresses security. It is motivated by the requirements for access control and confidentiality in an event-based middleware. We argued that security is a crucial factor that determines the wide-spread adoption of a middleware platform. The security service follows a secure publish/subscribe model that ties role-based access control to event types. It consists of three main parts: a mechanism for boundary access control of event clients that use the middleware, different levels of trust in event brokers specifying their access rights to event types, and the encryption of attributes in event publications to ensure confidentiality. We argued that our secure publish/subscribe model has the advantage of using a single shared infrastructure that can accommodate the security needs of different clients, as opposed to having an inefficient physical separation of the overlay broker network into distinct trust domains. We also described cryptographic key management and how the service interoperates with Hermes.

9.2 Further Work

There are a wide range of potential research avenues in which an event-based middleware, such as Hermes, can be extended. In general, more intelligence can be added to the middleware layer, which then takes over data processing responsibilities in addition to pure communication tasks. Composite event detection is already a step in this direction but event-based middleware can benefit further from a tighter integration with database systems. Similar to the CORBA object services in Section 2.1.1, the event-based middleware can be extended with a variety of additional services. For a large-scale deployment, a naming and directory service is necessary to support the global querying and browsing of event type repositories stored at rendezvous nodes. Next we list three interesting research challenges for an event-based middleware.

Reliable Delivery Semantics. The reliability model for an event-based middleware in Section 3.3.4 demands guaranteed delivery semantics for events. However, a challenging problem is how to implement persistent events efficiently so that the overall scalability of the system does not suffer. The guaranteed delivery service of Gryphon (see Section 2.2.2) reliably disseminates events but relies on a rather static overlay broker topology, so integration of reliable delivery semantics with robust peer-to-peer routing using dynamic topologies still remains as research to be investigated.

Mobility Support. Support for mobile clients is important in mobile ad-hoc networks and event-based middleware is a promising platform in these environments because of its loose coupling between components. To support the roaming of event clients, peer-to-peer event routing algorithms must be adaptable to changes in the location of event clients, thus using an even more dynamic notion of event dissemination trees. The desire of event clients to have durable subscriptions, which allow the selective replay of events missed during times of disconnected operation, motivates the addition of a query language that also enables event clients to subscribe to past events. This brings event-based middleware closer to database systems, thus unifying communication and storage.

Transaction Support. Transaction support has been proposed for message-oriented middleware [TR00, LT00]. Event-based middleware can also benefit from transactions because they allow stronger event dissemination semantics. Routing algorithms must observe the state of transactions while disseminating events. For example, the rollback of a primitive event publication can undo the detection of composite events at a detector. In addition, new types of transaction semantics are necessary to handle many-to-many communication in an event-based middleware.

Bibliography

[AADS02] Yair Amir, Baruch Awerbuch, Claudiu Danilov, and Jonathan Stanton. Global Flow Control for Wide Area Overlay Networks: A Cost-Benefit Approach. In *Proceedings of OPENARCH'02*, pages 155–166, June 2002.

[ABKM01] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[ACH$^+$01] Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a Sentient Computing System. *IEEE Computer Magazine*, 34(8):50–56, August 2001.

[Apa01] The Apache Project. Xerces Java Parser 1.4.0 Release. `http://xml.apache.org/xerces-j/`, June 2001.

[Apa03] The Apache Project. Xalan XSLT Processor. `http://xml.apache.org/xalan-j/`, 2003.

[ASS$^+$99] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of the 18th ACM Symposium on the Principles of Distributed Computing (PODC'99)*, Atlanta, GA, USA, May 1999.

[BBHM95] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using Events to Build Distributed Applications. In *Proceeding of the IEEE Services in Distributed and Networked Environments Workshop (SDNE'95)*, pages 148–155, Whistler, BC, Canada, June 1995.

[BCA$^+$01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, et al. The Design and Implementation of Open ORB Version 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[BCF94] Anastasios T. Bouloutas, Seraphin Calo, and Allan Finkel. Alarm Correlation and Fault Identification in Communication Networks. *IEEE Transactions on Communications*, 42(2/3/4):523–533, February 1994.

[BCM87] Rajive L. Bagrodia, K. Mani Chandy, and Jayadev Misra. A Message-Based Approach for Discrete-Event Simulation. *IEEE Transactions on Software Engineering (TSE)*, 13(6):654–665, June 1987.

[BCM$^+$99] 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 *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 262–272, Austin, TX, USA, May 1999.

[BCSS99] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A Case for Message Oriented Middleware. In Prasad Jayanti, editor, *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, volume 1693 of *LNCS*, pages 1–18, September 1999.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[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 H. Arno Jacobsen, editor, *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003. ACM.

[BFC93] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core Based Trees (CBT). In *Proceedings of ACM SIGCOMM'93*, San Francisco, CA, USA, September 1993.

[BH03] Jean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley, 2003.

[BHM+00] Jean Bacon, Alexis Hombrecher, Chaoying Ma, Ken Moody, and Walt Yao. Event Storage and Federation using ODMG. In *Proceedings of the 9th International Workshop on Persistent Object Systems (POS9)*, pages 265–281, Lillehammer, Norway, September 2000.

[BKS+99] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information Flow-based Event Distribution Middleware. In *Proceedings of the Workshop on Electronic Commerce and Web-Based Applications. In conjunction with the International Conference on Distributed Computing Systems (ICDCS'99)*, pages 114–122, Austin, TX, USA, May 1999.

[BMB+00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, pages 68–77, March 2000.

[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.

[BOP94] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of ACM SIGCOMM'94*, London, United Kingdom, August 1994.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 183–200, Vancouver, BC, Canada, October 1998.

[BP96] Lawrence S. Brakmo and Larry L. Peterson. Experiences with Network Simulation. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, pages 80–90, Philadelphia, PA, USA, May 1996.

[BSB+02] Sumeer Bhola, Robert Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 7–16, Washington, D.C., USA, June 2002.

[BZA03] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably Supporting Durable Subscriptions in a Publish/Subscribe System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, San Francisco, CA, USA, June 2003.

[Car98] Antonio Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-Area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, December 1998.

[CBB+97] Rick G. G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[CBP+02] Jon Crowcroft, Jean Bacon, Peter Pietzuch, George Coulouris, and Hani Naguib. Channel Islands in a Reflective Ocean: Large-scale Event Distribution in Heterogeneous Networks. *IEEE Communications Magazine*, 40(9):112–115, September 2002.

[CDHR02] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting Network Proximity in Distributed Hash Tables. In Ozalp Babaoglu, Ken Birman, and Keith Marzullo, editors, *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages 52–55, Bertinoro, Italy, June 2002.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 3rd edition, 2001.

[CJK$^+$03] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-Peer Overlays. In *Proceedings of INFOCOM'03*, San Francisco, CA, USA, April 2003.

[Cla99] Ian Clarke. A Distributed Decentralised Information Storage and Retrieval System. Master's thesis, University of Edinburgh, 1999.

[CM93] Sharma Chakravarthy and Deepak Mishra. Snoop — An Expressive Event Specification Language For Active Databases. Technical Report UF-CIS-TR-93-007, Department of Computer and Information Sciences, University of Florida, March 1993.

[CMB00] Yatin Chawathe, Steven McCanne, and Eric A. Brewer. RMX: Reliable Multicast for Heterogeneous Networks. In *Proceedings of INFOCOM'00*, pages 795–804, Tel Aviv, Israel, March 2000.

[CN01] Gianpaolo Cugola and Elisabetta Di Nitto. Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Proceedings of Middleware for Mobile Computing Workshop. In Conjunction with Middleware'01*, Heidelberg, Germany, November 2001.

[CNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI Event-Based Infrastructure and its Applications to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering (TSE)*, 27(9):827–850, September 2001.

[Cou01] Geoff Coulson. What is Reflective Middleware? *IEEE Distributed Systems Online*, 2(8), 2001. `http://computer.org/dsonline/middleware/RMarticle1.htm`.

[CP02] Jon Crowcroft and Ian Pratt. Peer to Peer: Peering into the Future. In *Advanced Lectures on Networking, NETWORKING 2002*, volume 2497 of *LNCS*, pages 1–19. Springer Verlag, 2002.

[CRW99] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Proceedings of Engineering Distributed Objects (EDO'99)*, Los Angeles, CA, USA, May 1999.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-Based Addressing and Routing: A General Model and its Application. Technical Report CU-CS-902-00, University of Colorado, Department of Computer Science, January 2000.

[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*, 19(3):332–383, August 2001.

[CRZ00] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A Case for End System Multicast. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)*, pages 1–12, Santa Clara, CA, USA, June 2000.

[CST01] Brad Cain, Tony Speakman, and Don Towsley. Generic Router Assist (GRA) Building Block Motivation and Architecture. Internet Draft, IETF, July 2001.

[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*, volume 2538 of *LNCS*, pages 59–68, Scottsdale, AZ, USA, October 2001.

[Dan89] Peter B. Danzig. *Optimally Selecting the Parameters of Adaptive Backoff Algorithms for Computer Networks and Multiprocessors.* PhD thesis, University of California, Berkeley, 1989.

[DBC96] Umeshwar Dayal, Alejandro P. Buchmann, and Sharma Chakravarthy. The HiPAC Project. *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 177–206, 1996.

[Dee89] Steve Deering. *Host Extensions for IP Multicast (RFC 1112).* Internet Engineering Task Force (IETF), 1989.

[DM78] Yogen K. Dalal and Robert M. Metcalfe. Reverse Path Forwarding of Broadcast Packets. *Communications of the ACM*, 21(12):1040–1047, December 1978.

[DO97] Dante DeLucia and Katia Obraczka. Multicast Feedback Suppression Using Representatives. In *Proceedings of INFOCOM'97*, pages 463–470, Kobe, Japan, April 1997.

[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.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian H. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, pages 131–146, Tampa, FL, USA, October 2001.

[EGH+03] Patrick Th. Eugster, Rachid Guerraoui, Sidath Handurukande, Rachid Guerraoui, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight Probabilistic Broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, November 2003.

[EGS00] Patrick Th. Eugster, Rachid Guerraoui, and Joe Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *Proceedings of the 14th AITO European Conference on Object Oriented Programming (ECOOP'00)*, pages 252–276, Sophia Antipolis and Cannes, France, June 2000.

[Eri94] Hans Eriksson. MBONE: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[Eug01] Patrick Th. Eugster. *Type-Based Publish/Subscribe.* PhD thesis, EPFL Lausanne, Lausanne, Switzerland, 2001.

[FF99] Sally Floyd and Kevin Fall. Promoting the Use of End-to-end Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[FIP95] FIPS 180-1. *Secure Hash Standard.* National Institute of Standards and Technology (NIST), April 1995. Federal Information Processing Standard (FIPS).

[FJG+98] Sonia Fahmy, Raj Jain, Rohit Goyal, Bobby Vandalore, Shivkumar Kalyanaraman, Sastri Kota, and Pradeep Samudraand. Feedback Consolidation Algorithms for ABR Point-to-Multipoint Connections in ATM Networks. In *Proceedings of IEEE INFOCOM'98*, volume 3, pages 1004–1013, San Francisco, CA, USA, March 1998.

[FJL+01] Francoise Fabret, Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth Ross, and Dennis Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proceedings of the 20th International Conference on Management of Data (SIGMOD'01)*, pages 115–126, Santa Barbara, CA, USA, May 2001.

[FKM+02] Geraldine Fitzpatrick, Simon Kaplan, Tim Mansfield, Arnold David, and Bill Segall. Supporting Public Availability and Accessibility with Elvin: Experiences and Reflections. *Computer Supported Cooperative Work*, 11(3):447–474, 2002.

[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.

[FMG03] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular Event-based Systems. *The Knowledge Engineering Review*, 17(4):55–85, 2003.

[FMMB02] Ludger Fiege, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering Event-based Systems with Scopes. In B. Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *LNCS*, pages 309–333, Malaga, Spain, June 2002.

[FP01] Sally Ford and Vern Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking (TON)*, 9(4):392–403, August 2001.

[GD94] Stella Gatziu and Klaus R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-AIDS'94)*, pages 2–9, February 1994.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GJS92] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Event Specification in an Active Object-Oriented Database. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'92)*, pages 81–90, San Diego, CA, USA, June 1992.

[Gon02] Li Gong. Project JXTA: A Technical Overview. Whitepaper, Sun Microsystems, October 2002. `http://www.jxta.org`.

[GS99] S. Jamaloddin Golestani and Krishan K. Sabnani. Fundamental Observations on Multicast Congestion Control in the Internet. In *Proceedings of INFOCOM'99*, pages 990–1000, New York, NY, March 1999.

[Hay96] Richard Hayton. *OASIS: An Open Architecture for Secure Interworking Services.* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, June 1996. Technical Report No. 399.

[HBWP01] Jie Huang, Andrew Black, Jonathan Walpole, and Calton Pu. Infopipes — An Abstraction for Information Flow. In *Proceedings of the ECOOP Workshop on The Next 700 Distributed Object Systems*, Budapest, Hungary, June 2001.

[HKRZ02] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, pages 41–52, Winnipeg, Canada, August 2002.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[Hom02] Alexis B. Hombrecher. *Reconciling Event Taxonomies Across Administrative Domains.* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, June 2002.

[IBM01] IBM TJ Watson Research Center. Gryphon: Publish/Subscribe over Public Networks. `http://researchweb.watson.ibm.com/gryphon/Gryphon`, December 2001.

[IBM02a] IBM Corporation. IBM WebSphere MQ. `http://www.ibm.com/software/integration/wmq/`, April 2002.

[IBM02b] IBM Corporation. IBM WebSphere MQ Event Broker. `http://www.ibm.com/software/integration/mqfamily/eventbroker/`, May 2002.

[ITU00] ITU-T. ITU-T X.509. Recommendation, ITU-T International Telecommunication Union, Geneva, Switzerland, 2000.

[JGJ+00] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*, San Diego, CA, USA, October 2000.

[JK88] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM'88*, pages 314–332, Stanford, CA, USA, August 1988.

[JS03] Yuhui Jin and Rob Strom. Relational Subscription Middleware for Internet-Scale Publish-Subscribe. In H. Arno Jacobsen, editor, *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1):41–50, January 2003.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The Case for Reflective Middleware. *Communication of the ACM*, 46(6):33–38, June 2002.

[KDHP03] Evangelos Kotsovinos, Boris Dragovic, Steven Hand, and Peter R. Pietzuch. XenoTrust: Event-Based Distributed Trust Management. In *Proceedings of Trust and Privacy in Digital Business (TrustBus'03). In conjunction with the 14th International Conference on Database and Expert Systems Applications (DEXA'03)*, Prague, Czech Republic, September 2003.

[KRL$^+$00] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Caludio Magalhaes, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Middleware (Middleware'00)*, volume 1795 of *LNCS*, pages 121–143, New York, NY, USA, April 2000.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LCB99] Christoph Liebig, Mariano Cilia, and Alejandro P. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the 4th International Conference on Cooperative Information Systems (COOPIS'99)*, pages 70–78, Edinburgh, Scotland, September 1999.

[LT00] Christoph Liebig and Stefan Tai. Advanced Transactions. In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO'00)*, Davis, CA, USA, November 2000.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*, pages 147–155, Orlando, FL, USA, October 1987.

[MB98] Chaoying Ma and Jean Bacon. COBEA: A CORBA-Based Event Architecture. In Joe Sventek, editor, *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 117–132, Santa Fe, NM, USA, 1998.

[MF99] Steven McCanne and Sally Floyd. UCB/LBNL/VINT Network Simulator - ns (Version 2). `http://www.isi.edu/nsnam/ns/`, April 1999.

[MFB02] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS'02)*, volume 2299 of *LNCS*, pages 224–238, Karlsruhe, Germany, April 2002.

[MFGB02] G. Mühl, L. Fiege, F. Gärtner, and A. Buchmann. Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, pages 167–176, Fort Worth, TX, USA, October 2002.

[Mik02] Zoltán Miklós. Towards an Access Control Mechanism for Wide-area Publish/Subscribe Systems. In *Proceedings of the First International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, July 2002.

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of MASCOTS'01*, pages 346–356, Cincinnati, OH, USA, August 2001.

[MSS97] Masoud Mansouri-Samani and Morris Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[Müh02] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany, September 2002.

[Nir03] Anusha Nirmalananthan. A GUI Application for Composing Event Streams. Computer Science Tripos Part II Project Dissertation, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, May 2003.

[Obj02] ObjectWeb Open Source Middleware. JORAM Java Open Reliable Asynchronous Messaging 3.2.0 Release. `http://www.objectweb.org/joram`, October 2002.

[OMG95] OMG. CORBA: Event Service, Version 1.0. Specification, Object Management Group (OMG), March 1995.

[OMG02a] OMG. CORBA: Notification Service, Version 1.0.1. Specification, Object Management Group (OMG), August 2002.

[OMG02b] OMG. The Common Object Request Broker Architecture: Core Specification, Revision 3.0. Specification, Object Management Group (OMG), December 2002.

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus — An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP'93)*, pages 58–68, Asheville, NC, USA, December 1993.

[Ora01] Andrew Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, 1st edition, March 2001.

[PB02] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In Jean Bacon, Ludger Fiege, Rachid Guerraoui, H. Arno Jacobsen, and Gero Mühl, editors, *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02). In conjunction with the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 611–618, Vienna, Austria, July 2002. IEEE.

[PB03a] Peter R. Pietzuch and Jean Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In H. Arno Jacobsen, editor, *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003. ACM.

[PB03b] Peter R. Pietzuch and Sumeer Bhola. Congestion Control in a Reliable Scalable Message-Oriented Middleware. In Markus Endler and Douglas Schmidt, editors, *Proceedings of the 4th International Conference on Middleware (Middleware'03)*, volume 2672 of *LNCS*, pages 202–221, Rio de Janeiro, Brazil, June 2003. ACM/IFIP/USENIX, Springer Verlag.

[PCM03] Gian Pietro Picco, Gianpaolo Cugola, and Amy L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'03)*, Providence, RI, USA, May 2003.

[PD99] Norman W. Paton and Oscar Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, March 1999.

[Pet77] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[PF03] Shrideep Pallickara and Geoffrey Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In Markus Endler and Douglas Schmidt, editors, *Proceedings of the 4th International Conference on Middleware (Middleware'03)*, volume 2672 of *LNCS*, pages 41–61, Rio de Janeiro, Brazil, June 2003.

[Pie00] Peter R. Pietzuch. An Event Type Compiler for ODL. Computer Science Tripos Part II Project Dissertation, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, June 2000.

[Pie02] Peter R. Pietzuch. Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems? 6th CaberNet Radicals Workshop, February 2002.

[Pow96] David Powell. Group Communication. *Communications of the ACM*, 39(4):50–97, April 1996.

[PPF+03] Shrideep Pallickara, Marlon Pierce, Geoffrey Fox, Yan Yan, and Yi Huang. A Security Framework for Distributed Brokering Systems. `http://www.naradabrokering.org`, 2003.

[PS02] Peter R. Pietzuch and Brian Shand. A Framework for Object-Based Event Composition in Distributed Systems. Presented at the 12th International Network for PhD Students in Object Oriented Systems (PhDOOS'02) Workshop. In conjunction with the 16th European Conference on Object-Oriented Programming (ECOOP'02), June 2002.

[PSB03] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A Framework for Event Composition in Distributed Systems. In Markus Endler and Douglas Schmidt, editors, *Proceedings of the 4th International Conference on Middleware (Middleware'03)*, volume 2672 of *LNCS*, pages 62–82, Rio de Janeiro, Brazil, June 2003. ACM/IFIP/USENIX, Springer Verlag.

[PSB04] Peter R. Pietzuch, Brian Shand, and Jean Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, 18(1):44–55, January/February 2004.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 3rd International Conference on Middleware (Middleware'01)*, pages 329–350, Heidelberg, Germany, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, USA, August 2001.

[RHKS01] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-Level Multicast Using Content-Addressable Networks. In Jon Crowcroft and Markus Hofmann, editors, *Proceedings of 3rd International Workshop on Networked Group Communication (NGC'01)*, volume 2233 of *LNCS*, pages 14–29, London, United Kingdom, November 2001.

[Riz00] Luigi Rizzo. pgmcc: A TCP-Friendly Single-Rate Multicast Congestion Control Scheme. In *Proceedings of ACM SIGCOMM'00*, Stockholm, Sweden, August 2000.

[RKCD01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The Design of a Large-Scale Event Notification Infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *LNCS*, pages 30–43, London, UK, November 2001.

[RLW+03] Anton Riabov, Zhen Liu, Joel L. Wolf, Philip S. Yu, and Li Zhang. New Algorithms for Content-Based Publication-Subscription Systems. In *Proceedings of 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 678–686, Providence, RI, USA, May 2003.

[Rob94] Lawrence Roberts. Rate-based Algorithm for Point to Multipoint ABR Service. ATM Forum Contribution 94-0772R1, November 1994.

[RW97] David S. Rosenblum and Alexander L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the 6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.

[SA97] Bill Segall and David Arnold. Elvin has left the Building: A Publish/Subscribe Notification Service with Quenching. In *Proceedings of AUUG Technical Conference '97*, Brisbane, Australia, September 1997.

[SAB+00] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content Based Routing in Elvin4. In *Proceedings of AUUG2K*, Canberra, Australia, June 2000.

[SAS01] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting Disconnectedness — Transparent Information Delivery for Mobile and Invisible Computing. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, Brisbane, Australia, May 2001.

[Sat96] Shirish S. Sathaye. ATM Forum Traffic Management Specification 4.0. ATM Forum af-tm-0056.000, April 1996.

[SCFY96] Ravi Sandhu, Edward Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[Sch96] Scarlet Schwiderski. *Monitoring the Behaviour of Distributed Systems*. PhD thesis, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, 1996.

[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 ACM SIGCOMM'01*, San Diego, CA, USA, August 2001.

[Spi00] Mark David Spiteri. *An Architecture for the Notification, Storage and Retrieval of Events*. PhD thesis, University of Cambridge Computer Laboratory, Cambridge, United Kingdom, January 2000.

[SQL92] X/Open CAE: Structured Query Language (SQL). Specification Data Management Version 2, The Open Group, March 1992.

[Sun99a] Sun Microsystems. Java Naming and Directory Interface (JNDI). Specification, Sun Microsystems, July 1999. `http://java.sun.com/products/jndi/`.

[Sun99b] Sun Microsystems. Java Remote Method Invocation (RMI). Specification, Sun Microsystems, 1999. `http://java.sun.com/products/jdk/rmi/`.

[Sun01] Sun Microsystems. Java Message Service. Specification, Sun Microsystems, 2001. `http://java.sun.com/products/jms/`.

[Sun03a] Sun Microsystems. JavaSpaces Service Specification, Version 2.0. Specification, Sun Microsystems, June 2003.

[Sun03b] Sun Microsystems. Jini Specification, Version 2.0. Specification, Sun Microsystems, June 2003. `http://java.sun.com/products/jini/`.

[SW00] Sherlia Shi and Marcel Waldvogel. A Rate-based End-to-end Multicast Congestion Control Protocol. In *Proceedings of 5th IEEE Symposium on Computer and Communication (ISCC'00)*, Antibes-Juan les Pins, France, July 2000.

[TIB99] TIBCO. TIBCO Rendezvous. `http://www.rv.tibco.com`, 1999.

[TR00] Stefan Tai and Isabelle Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In Joe Sventek and Geoff Coulson, editors, *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*, pages 308–330, Hudson River Valley, NY, USA, 2000.

[TSLK01] Puneet Thapliyal, Sidhartha, Jiang Li, and Shivkumar Kalyanaraman. LE-SBCC: Loss-Event Oriented Source-based Multicast Congestion Control. Technical report, Rensselaer Polytechnic Institute ECSE, 2001.

[TSWM97] David L. Tennenhouse, Jonathan M. Smith, W. David Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[W3C99a] W3C. Namespaces in XML. W3C Recommendation, World Wide Web Consortium, January 1999.

[W3C99b] W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation, World Wide Web Consortium, November 1999.

[W3C01a] W3C. XML Schema Part 0: Primer. W3C Recommendation, World Wide Web Consortium, May 2001.

[W3C01b] W3C. XML Schema Part 1: Structures. W3C Recommendation, World Wide Web Consortium, May 2001.

[W3C01c] W3C. XML Schema Part 2: Datatypes. W3C Recommendation, World Wide Web Consortium, May 2001.

[W3C03a] W3C. SOAP Version 1.2 Part 0: Primer. W3C Recommendation, World Wide Web Consortium, June 2003.

[W3C03b] W3C. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, World Wide Web Consortium, June 2003.

[W3C03c] W3C. SOAP Version 1.2 Part 2: Adjuncts. W3C Recommendation, World Wide Web Consortium, June 2003.

[W3C03d] W3C. XQuery 1.0: An XML Query Language. W3C Working Draft, World Wide Web Consortium, November 2003.

[WCEW02] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L. Wolf. Security Issues and Requirements in Internet-scale Publish-subscribe Systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, page 303, Big Island, HI, USA, 2002.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective Dynamics of Small-World Networks. *Nature*, 393:440–442, June 1998.

[Wyc98] Peter Wyckoff. T-Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[YHF$^+$03] Yan Yan, Yi Huang, Geoffrey Fox, Shrideep Pallickara, Marlon Pierce, Ali Kaplan, and Ahmet Topcu. Implementing a Prototype of the Security Framework for Distributed Brokering Systems. In *Proceedings of the International Conference on Security and Management (SAM'03)*, pages 212–218, Las Vegas, NV, USA, June 2003.

[YL00] Yang Richard Yang and Simon S. Lam. Internet Multicast Congestion Control: A Survey. In *Proceedings of the International Conference on Telecommunications (ICT'00)*, Acapulco, Mexico, May 2000.

[ZCB96] Ellen Zegura, Kenneth Calvert, and Samrat Bhattacharjee. How to Model an Internetwork. In *Proceedings of INFOCOM'96*, pages 594–602, San Francisco, CA, USA, 1996.

[ZF01] Daniel Zappala and Aaron Fabbri. An Evaluation of Shared Multicast Trees with Multiple Active Cores. In *Proceedings of the IEEE International Conference in Networking (ICN'01)*, pages 620–629, Colmar, France, July 2001.

[ZKJ01] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical report, Computer Science Division, University of California, Berkeley, Berkeley, CA, USA, April 2001.

[ZSSK02] Xi Zhang, Kang G. Shin, Debanjan Saha, and Dilip D. Kandlur. Scalable Flow Control for Multicast ABR Services in ATM Networks. *IEEE/ACM Transactions on Networking*, 10(1), February 2002.

[ZZJ$^+$01] Shelley Q. Zhuang, Ben Y. Zhao, Anthony Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the 11th International Workshop on Network and OS Support for Digital Audio and Video (NOSSDAV'01)*, June 2001.