Reconciling Event Taxonomies Across Administrative Domains

Alexis B. Hombrecher

Jesus College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

June 2002

DECLARATION

Except where otherwise stated in the text, this thesis is the result of my own work and is not the outcome of work done in collaboration.

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.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

ACKNOWLEDGEMENTS

Ken Moody, my supervisor, has been a tremendous help and has been very supportive throughout my time in Cambridge, particularly during the last couple of months of my PhD. His insights and advice have been invaluable.

I also want to thank Jean Bacon who has always been very supportive and very generous, even in times when I pursued the odd extra-curricular activity.

I am grateful to all the members of the OPERA Group who have made my time in Cambridge very special. A special *thanks* also goes to all my colleagues with whom I have had very helpful and inspiring discussions, particularly Walt Yao and Peter Pietzuch.

My research was supported by the Engineering and Physical Science Research Council (EPSRC). I want to thank them for their help.

I want to also thank my parents who have provided me with so many opportunities to enable me to be where I am today. Their continued support and help have opened many doors for me.

Finally, I want to thank Erin, without whose help I would have probably never been able to start writing this thesis. A motivational talk on a cross-country drive got me in a writing mood. She has been the biggest help and greatest inspiration anyone could hope for.

ABSTRACT

Event notification systems have proved to be a well suited architecture for linking heterogeneous applications in a loosely coupled and flexible manner. Internet-scale event notification systems have taken this approach a step further with applications not just communicating at a local-area network level, but on an internet-wide level.

In the database world sharing heterogeneous information across platforms has been a research topic for over twenty-five years. Database federation has developed a great deal in this time, making heterogeneous database integration and information sharing a reality. Event notification systems have this same need to share heterogeneous information. Sharing information becomes a requirement not only across heterogeneous applications, but also across heterogeneous event notification and messaging systems.

Currently, no flexible model is available with the capabilities to link existing event notification architectures. This thesis proposes an event federation paradigm to make information sharing across heterogeneous event systems a reality. The paradigm is based on many of the ideas developed in database federation research.

This thesis begins by describing existing event architectures. It traces the use of meta-data, to describe event systems and their event types, in middleware architectures. This is followed by an overview of federation research in databases. Here we look at the classical reference architecture as well as the wrapper/mediator model. Based on these ideas the thesis develops a paradigm to link event notification systems. The main components in this architecture are domains, gateways and contracts. Domains establish contracts with each other. The contracts define how domains share information. Gateways manage the communication between domains, maintaining contracts and meta-functions to perform event translations according to the details of a contract.

An XML architecture for gateways based on HERMES, a content-based event notification system, is described. This architecture is then used in an experimental framework. The first is the *Active Street*, consisting of modern houses. These are *Active Houses*, whose appliances and other components are connected via an event architecture. This idea is extended and an *Active Street* consisting of heterogeneous *Active Houses* is constructed using the event federation architecture. We use the same principles and show how *Active Cities* can be built using the event federation paradigm. The second experiment uses the XML-based architecture to connect heterogeneous mobile devices.

Finally, the thesis explores possible extensions to the event federation architecture. The focus will be on enhancements which might make it possible to incorporate other types of middleware into the paradigm.

LIST OF FIGURES

CONTENTS

1. INTRODUCTION

Imagine your mobile communicator notifying you that your children have just arrived home, while you are still standing in a queue waiting for a taxi to take you home. Fortunately you are able to monitor the arrival times of the next ten taxis on a large public LCD screen, thus informing you that a taxi will be there for you in approximately seven minutes. Using the real-time traffic information system on your mobile device you are able to determine that it will take you exactly thirteen minutes to get home from the time the taxi picks you up. Using your mobile communicator you are able to prepare the evening meal by boiling water and turning on the oven, checking whether the supermarket has delivered the necessary groceries, which it does automatically since the intelligent refrigerator notifies the supermarket when groceries have to be delivered, and call the children to inform them that you will be home in twenty minutes.

Some of the technology described above is already available today. In Helsinki, for example, Hewlett Packard is testing a public tram and bus monitoring system. This allows people to monitor bus and tram arrival times in real-time on their mobile phones. When the bus or tram is near the desired stop, the system sends a text message to all subscribers of the service. A bus monitoring application using mobile phone technology has also been developed and tested in Dublin [Fal00]. Besides transportation monitoring systems, the concept of an *Active House* is also becoming a reality. In such a house appliances are connected to a network and are capable of communicating intelligently with each other. A motion detection system might spot the presence of a human opening a door and as a result automatically activate the lights and music in the room.

The communication model connecting the different applications and devices in the above scenarios, including the mobile communicator, the home appliances, the public taxi monitor, the bus monitoring system and the real-time traffic information system is event-based middleware. Event-based middleware has emerged as the most suitable means of connecting distributed applications in a heterogeneous environment where central control is not desirable and asynchronous communication is essential. It is ideally suited for constructing component-oriented systems that monitor and react to changes in the environment.

We are envisioning a world of active buildings and cities, with many roving entities and devices generating a potentially large volume of events, and (mobile) users and applications interested in receiving them. It is unlikely that the world will restrict itself to one single event platform to allow global communication between the devices as it was described above. Middleware was developed because of the need to connect heterogeneous applications. There is a similar need to connect heterogeneous middleware platforms, specifically event-based middleware platforms. This thesis explores and describe a generic approach for the inter-operation of event-based systems.

1.1 Event-based Middleware

Many different flavors of event-based middleware exist, focusing on a variety of aspects such as reliability, scalability, usability and extensibility. Although these systems all differ slightly, they share many common aspects. The aim of any event-based architecture is to connect producers of information to entities interested in that information. Information is modeled in the form of events, where an event denotes that which arises from a preceding state of things. An event is likely to have distinguishing attributes or properties. For example, an event about a change in a stock price may have properties such as *stock name*, *change* and *current value*. These properties allow a user to distinguish different *stock price change* events from one another. The concept of an event is central to event-based middleware. In its simplest form, an event is an occurrence of some change of state within our universe. Different examples of events are:

- a moving entity, such as a person entering a room in a building
- a change to a data value, such as the market price of a stock
- a program exception, such as an exponent overflow during a calculation
- a particular time of day during the week, such as 6 pm on a Friday
- a change to a tuple in a table of a database that violates an integrity constraint
- the drawing of a line on a multi-user interactive white board, which needs to be shown on all participating white boards
- a change in the computing environment, such as a drop in available bandwidth
- an external physical change, such as the temperature dropping below freezing

The terminology used in event-based systems varies greatly, yet the underlying concepts are very similar. In these systems events, messages, notifications or announcements serve as the glue to connect programs, applications, modules, processes, clients, objects or components.

Most often, event-based middleware is described as a *publish/subscribe* or *publish/register/notify* system. A component generating events is known as a *source*, while the consumer of events is known as a *sink*. Sources publish the events they generate, while sinks register interest with particular sources, for certain event types. Upon the occurrence of an event, interested sinks are notified.

If an application is to adapt to the occurrence of events, it must be able to detect them. For this, events need to be classified, so that each event occurrence can be reported to the appropriate client. In instances where events are not classified, clients register interest with a service and receive all event occurrences generated by the service. This is clearly not efficient, using up valuable network resources. Event classification is one of the keys to *source-side filtering* and as a result reduced network traffic.

An important aspect of events is that they can occur at any time, outside the control of interested parties. Furthermore, it is hard to know who and how many parties might be interested in a particular event. To handle this aspect, event-based communication occurs asynchronously, extending existing synchronous middleware platforms. This allows the senders and receivers of notifications to be decoupled not only in space, but also in time. Unlike with direct method invocation models the sender does not have to wait for a receipt from the client, but can continue with other duties. Delivery guarantees are generally handled by the architecture rather than the source of the event.

1.2 Applications of Event-based Middleware

Event-based middleware has found wide-ranging application in computing. It is used in building and composing large, scalable distributed systems, enterprise application integration, monitoring and measurement-taking applications, real-time sensor based environments, active databases, windowing systems, programming languages to decouple communication between concurrent objects or modules and distributed collaborative shared platforms. This list is not exhaustive and only gives an indication of the paradigms applicability.

More generally, event-based systems are well suited for connecting heterogeneous applications in a loosely coupled manner and for applications that require awareness support. Specific applications of these two general areas are listed below.

Alarms and Exceptions These are notified without delay. Synchronous polling is an unacceptable means of detection.

System Management This includes monitoring and control, for example for timely fault detection or report and repair in a telecommunications network. Monitoring by polling is inadequate.

Active Databases An active database uses events to monitor storage operations and can carry out specific actions upon an occurrence of certain conditions pertaining to the state of its data [MD89, CKAK94, KL98, Buc99].

Group Interaction Changes to copies of shared objects must be communicated promptly to all participants.

Multimedia Support When controlling an authored, interactive multimedia presentation, the user may pause the presentation, put up supplementary information, then continue. The presentation is driven by the user's interaction events [Bat96].

Sensor-based Environments For example, a bus is detected by a specific sensor in an active city. A transportation service is notified of all such events and updates displays at the bus station [5T02, DM01, DWMC00].

Support for Mobile Entities Network-enabled devices such as palmtops and telephones will be able to act as event sources and sinks.

From these descriptions, three main reasons that had a major impact on the growth and development of event-based middleware can be identified. The first is the need for heterogeneous application integration in a loosely coupled manner for building large, distributed systems. The second is reactive systems which require event notification to trigger actions within an application. The third area is awareness support in collaborative applications [Pri99, RDR98] where events coordinate between multiple users or entities of a system. This area includes applications such as sensor-based systems that monitor the whereabouts of entities within an environment, multi-user games and computer-supported collaborative work environments.

Currently, most event-based applications communicate at a local-area network level at best. This will change and soon it will be a requirement to link information producers and consumers at an internet-scale. It is unrealistic to assume that the world will adopt a single standard for event-based middleware [GHM96]. It is much more likely that existing event-based architectures need to be integrated in an efficient and scalable manner. We have already seen this with databases where much research has been undertaken in the hope of finding a plausible and efficient way of connecting multiple heterogeneous databases. Similar to the idea of connecting heterogeneous databases in a federated manner, we propose federations for connecting event systems. Here, individual, heterogeneous event architectures are able to communicate and exchange event information in a loosely coupled manner.

1.3 Why Event Federations?

The notion of federations is not uncommon, both in the real world and in computing. In computing, federations are used to link heterogeneous database systems or information sources. The need to link databases for information sharing has not subsided and will continue to be an important research area. The same will be true for event-based systems and middleware in general.

Above we have described a tracking system for mobile entities as an application requiring event-driven notification. This application lends itself well for describing not only the concept, but also the need for event federation. Many different types of tracking systems exist. Currently they include electronic badge-based tracking systems [WHFG92, HH94], login-based systems or systems using the Global Positioning System (GPS). At the moment they are all closed systems, meaning they can only track entities within their specific application domain. The electronic badge-based system, for example, can only monitor the location of entities where hardware for badge-based tracking is installed. Once an entity leaves this tracking domain, the system can no longer locate the electronic badge. For certain application domains, this type of tracking is sufficient, but we argue that soon the need for different tracking domains to communicate with each other will arise.

In a federated event architecture, a client can register with a tracking service in a given application domain. The client is not only notified of event occurrences from the local application domain, but also from other application domains which are part of the federation. An entity can be tracked wherever tracking system domains are installed and can communicate with each other.

Interestingly, very few middleware applications focus on the interoperability between middleware. Even though DCOM/CORBA bridges [RCF98] have been developed in recent years and due to the standardization efforts by OMG, different CORBA ORBs are able to communicate even if they come from different vendors, no general solutions exist that make middleware integration a reality. With more specialized middleware applications, such as event-based or message-oriented middleware, interoperability is even less common.

1.4 Research Aims

The aim of this dissertation is to develop and describe an architecture for connecting heterogeneous event-based middleware. This requires services to overcome both semantic and system heterogeneity. An event architecture must provide these additional services to join a federation. Besides a global data model for inter-domain communication, a global subscription language is required for cross-domain subscriptions. Finally, a paradigm for sharing heterogeneous information across component event systems is needed.

This thesis describes a novel approach for linking heterogeneous event systems. The approach combines new ideas with those developed in federated database and event-based middleware research. The new ideas developed and described in this thesis are listed below:

An Event Federation Paradigm Linking heterogeneous event platforms has not been a goal of existing architectures. A paradigm is needed to describe what an event federation is and what its requirements are. Generally, linking systems requires that they agree on a communication protocol. The paradigm describes this protocol.

An Event Federation Architecture Using the event federation paradigm, an architecture for linking existing event platforms is discussed. A global data model for the federation and a global subscription language for cross-domain subscription is described. The components that resolve semantic and system heterogeneity are discussed in detail.

An Event Federation Application The federated architecture is tested in applications linking heterogeneous event architectures. This tests both the paradigm and the architecture.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 discusses the background information necessary to develop the notion of event federation. It begins by looking at the history of middleware. It then looks at different event architectures and messaging systems and shows how the concept of typed events has developed within these systems. It illustrates the importance of metadata in these architectures for such things as event storage and content-based routing.

Chapter 3 discusses the concept of federations, specifically focusing on database federation, a problem which has been studied for the past three decades. It begins by looking at a reference architecture for database federation. Following that, different types of implementations are discussed, paying particular attention to wrapper/mediator technology. Finally, the chapter looks at the specific types of heterogeneity that are encountered in system integration.

Chapter 4 introduces the event federation paradigm central to this work. It begins by defining and explaining the necessary concepts. Specifically, it explains what a federation is and what the requirements are for joining a federation; it explains the concept of an event domain and outlines the requirements for a domain to join a federation; it discusses the notion of a contract and how these are used to link domains and integrate them in a federated style. Finally, the chapter explains the concept of gateways and describes their functionality.

Chapter 5 discusses an XML-based event federation architecture. It begins by giving an overview and describing the tools used to implement the architecture. It discusses the data definition language and global subscription language used in the federation. It then describes HERMES, a content-based messaging platform based on XML, used to construct the inter-domain communication layer of gateways. The chapter then focuses on the translation modules of a gateway, specifically those components managing event and subscription translation. Finally, technical aspects involving contract and global type creation are discussed.

Chapter 6 describes experiments and case studies showing how our federated event architecture is used. We look specifically at the *Active House* and *Active City* environments. Furthermore, we discuss scenarios in mobile computing for which event federation is well suited.

Chapter 7 analyzes the limitations of the event federation paradigm and its implementation. It critically evaluates the federated architecture described in this thesis and makes suggestions for further research directions which may help to improve the system.

Chapter 8 gives a brief conclusion, summarizing the work described in this thesis.

2. EVENT-BASED MIDDLEWARE

The focus of this chapter is on event-based middleware platforms and their evolution from simple topic-based messaging systems to internet-scale event notification systems. Special attention is paid to the notion of typing in these systems. By describing and evaluating the spectrum of event-based middleware, some of the requirements for federating event systems are highlighted.

Event-based systems are a specific type of middleware, which have evolved in the last decade. The aim of event-based systems is to connect heterogeneous applications, support reactive systems or provide awareness support by sending notifications of an occurrence of some event to interested entities. Entities indicate their interest in events by registering with a service, either directly or indirectly. The service notifies entities when an event of interest occurs. Besides this basic functionality, event-based middleware may also supports a range of other services such as event storage, transaction support or naming services, just to name a few. These services are implementation specific. We describe some of these implementations, specifically focusing on aspects which are relevant for federating event systems.

Section 2.1 gives a brief overview of the different types of middleware platforms and coupling models. We look at their development over the past two decades. This provides a context for describing event-driven systems. In section 2.2 the focus is on the different types of event-based middleware, specifically the suitability of different systems for event federation. Finally, in section 2.3 we review architectures developed for internet-scale event notification, also known as content-based routing systems.

2.1 Distributed Middleware

The term *middleware* first appeared in the 1980s and was used to describe network connection management software. The term did not become popular until the mid 90s when network technology had finally matured and became more visible, partially due to the rapid growth of the internet. At that time middleware had developed into a

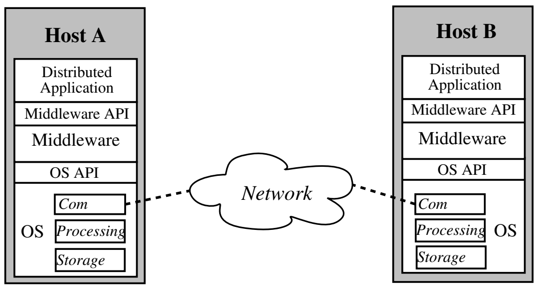


Fig. 2.1: General Middleware Architecture

much richer set of services and helped in building and managing distributed systems. The term today denotes software components running between application layer and operating system. It is a distributed software layer which abstracts over the complexity and heterogeneity of the underlying distributed, and possibly heterogeneous, environment. Middleware simplifies the task of designing and managing distributed applications. Its components must satisfy some basic requirements such as locating application processes distributed among several machines and allowing those to communicate, while resolving heterogeneity between different platforms wishing to communicate. Generally, distributed middleware tries to abstract away most aspects of distribution and heterogeneity, such as programming language and operating system heterogeneity. Figure 2.1 shows the concept of middleware in its most generic form. Applications use the API of the middleware architecture, which in turn uses the API of the operating system to connect to remote applications.

It is difficult to classify middleware and place an instance in a specific category. Their different characteristics often make them hybrids. Previous work has tried to loosely group middleware into categories such as client/server, distributed object, message-oriented, distributed tuple and event-driven. For the purpose of this dissertation, rather than categorizing distributed middleware it is more important to explain the different purpose each type of middleware serves and highlight their key characteristics. We therefore describe some of the different middleware platforms and elaborate on their functionality. The descriptions should not be viewed as a classification, but rather as a guideline and overview to give the reader a historic perspective on the emergence of event-based systems.

2.1.1 Early Middleware - The First Generation

In its infancy, middleware was mainly associated with database connectivity. It provided an interface for applications to connect to local, and later, remote database systems. We might call this technology database-oriented middleware. This form of middleware provided an API for an application to hide the operating system and network layer from the application. Today we have standards such as *ODBC* [Net], *JDBC* [FCHH99] and *OLE-DB* [Nor96], all database-oriented middleware that allow different applications to connect to a remote database. These are standardized platforms that most database vendors support, allowing applications written in different programming languages and running on different operating systems to connect to a variety of databases, again running on different platforms, without having to be reprogrammed or even recompiled.

A more generic paradigm evolved from the database-oriented one. The type of middleware models that emerged were Remote Procedure Call (RPC) based, often referred to as client/server. This is, as the name implies, based on a procedure-oriented model and uses connection-based communication. Applications invoke remote procedures that are located within another program, which may be located on another machine on a different platform. The procedure to be performed is invoked by the client on the server, where it is then executed. RPC invocation is synchronous with the control being passed from the local to the remote procedure. Local execution is blocked until a result is returned.

RPC systems usually enforce strong typing on the data being transferred since the client and server often use a common programming language. The server interface is mirrored on the client as a *stub* or *skeleton* which is linked with the client code to enforce the typing at compile time. Many proprietary RPC systems are currently in use, the main ones being *DCE's* RPC service [Fou92, RS93], *ONC* [Mic95], and *Java* Remote Method Invocation (*RMI*) [Sun].

2.1.2 The Second Generation

The second generation of middleware platforms began to emerge in the early 90s. The object-oriented paradigm in programming had become popular, thus making an object-based middleware platform the obvious choice. The focus was on distributed objects communicating with each other, once again hiding the communication details from the user. Although object-orientation was and still is a powerful paradigm and many of the concepts can still be found in today's middleware, these systems were not successful commercially. Around this time, the message-based paradigm emerged.

Message-Oriented Middleware (MOM) systems were a natural extension of the RPC-based systems. The greatest limitation of RPC-based systems was their synchronous communication model. MOM systems provide an asynchronous communication model using messages. This means that the two sides taking part in the communication can be decoupled not only in space, but also in time. A message-producing application delivers messages to its messaging middleware, which then transports the message to the correct application or location. Senders and receivers of messages are thus decoupled since primary acknowledgment of delivery is placing the message with the messaging middleware. No acknowledgment of delivery is returned to the sender by the receiving application. Generally, messages contain a structured collection of fields which are mapped to a machine-readable string. The mapping and extent of typing supported varies according to the messaging system, but generally the only support is for a very limited type system.

An extension of MOM systems was Message-Queuing Middleware (MQM). MQM systems allows for delivery guarantees in the face of network failure. MQM places messages in a queue, which can be queried by interested applications. If permitted, applications can then retrieve messages from a queue. This is analogous to the concept of electronic mail, where messages are placed in a mailbox and applications (email programs) retrieve messages. This concept was further developed into message brokering systems. Rather than delivering messages to a queue, messages are sent to a broker which can handle the *publish/subscribe* paradigm as well as message transformation and message composition.

Parallel to the development of MOM systems and their derivatives, emerged the distributed object model. Unlike MOM, the distributed object model is connection-based, which is generally synchronous. The main difference here is that the interaction is between objects. Operations are performed by invoking methods on a local object created by the distributed object system. Objects interact via an Object Request Broker (ORB). The ORB handles requests that an object makes of another object, hiding the actual communication and mechanism of locating the remote object across the network.

This framework usually enforces strong data-typing and unlike the client/server framework aims to integrate heterogeneous components with different type systems. A language independent type system is used, which language-specific type systems map to. In CORBA, for example, the Object Management Group's (OMG) *IDL* [OMG00] is used to describe objects in a language independent way. This is then compiled into programming language dependent *stubs* used by the application. Type checking is done at compile time. The main distributed object frameworks in use are DCOM/COM+ [Red97] by Microsoft and CORBA [Obj98] by OMG.

2.1.3 The Third Generation

Most recently, event-based middleware has emerged as a paradigm for booth loose integration of heterogeneous components and providing awareness support for applications. In event-based architectures components are linked and communicate with each other via events. Events provide a convenient way of representing the outside world. An event can be a change in temperature, a change in a stock price, a change of address or a mouse click within a window. All these occurrences are events that some entity, an application, an object or even a person may be interested in. In the event paradigm, there are three main entities. These are event producers, known as *sources*, event consumers, known as *sinks* and a *message transfer service*, which delivers event messages from sources to sinks. Event sinks register interest with event sources, specifying what they want to be informed about. This can either be a simple topic-based registration or more complex attribute-based registration. When an event of interest occurs at a source, it is delivered to all interested sinks via the message transfer service. Services, such as subscription, fault-tolerance/error-handling, message transformation and filtering are positioned somewhere between sources, sinks and the message transfer component.

An important aspect of event systems is the message or event transfer component that delivers messages from the producer to the consumer. Between the client and the server, different services such as event registration (subscription), filtering, error-handling and message transformation are performed. These services can take place in different parts of the system and be either distributed or centralized. Message transformation, for example, may take place at the source or sink, but it may also be performed at a mediator service lying between the two entities. Generally, two approaches can be identified from among the different types of systems. The first is *direct*, the second *indirect*. In the direct approach event sources and sinks must know about each other and communicate directly with one another. In the indirect approach, server and client do not know about each other. They communicate with one another via an event or message channel.

Whereas first and second generation middleware are based on tight coupling between the interacting entities, third generation event architectures are based on loose coupling.

Tight coupling generally means the entities wishing to communicate must know about each other's API, be online at the same time and communicate synchronously. The advantages of loose coupling are immediately apparent. Loose coupling is more suited to the dynamic nature of distributed systems. The main advantages are:

- entities need to have almost no knowledge of each other prior to communication
- entities can change dynamically without affecting the running system
- entities communicate asynchronously and do not need to be connected to the network at all times

Many different event architectures have emerged, focusing on different aspects. The focus has mainly been on genericity, security, reliability and scalability. Most interesting for event federation is the genericity of event systems. By genericity, we mean how general purpose an event system is. Most event notification systems are tailored toward a specific application area and thus make assumptions about the environment they are deployed in. A generic system is one that applies no semantic meaning to the events it communicates.

Early event systems do not have type-support and events or messages consist of a string or unstructured bytes. In these instances, the sender and receiver of events make an assumption about the information they transmit. Only with this assumption will the event clients be able to understand the information. A type system does not only guarantee that an event architecture is more generic, it is also a prerequisite for event federation. In the next section we discuss the development of event architectures with type systems.

2.2 Metadata in Event Systems

[RK98] has evaluated over one hundred different event-based architectures. Their claim is that internet-scale event notification will bring about a common standard for event architectures. This is an assumption that has been disproved many times in computer science. We believe it is a much more realistic assumption that the world will continue to see new systems evolve while different standards continue to do battle. Heterogeneity will always have to be dealt with, making a universally accepted middleware standard unlikely. The need for integration is the very reason for the development and rapid rise to prominence of middleware. We believe that the world will become more complex resulting in a greater need for integration. Metadata is the key for loose coupling between applications and systems in general. This section therefore focus on describing event systems that use metadata to describe their events.

Much like humans, applications or systems can only communicate with each other when speaking the same language or when they have a translator. We have shown that speaking the same language is a requirement for systems that are tightly coupled. In an environment where loose coupling is desired, another method for enabling heterogeneous systems to communicate is needed. This comes in the form of metadata. The most generic middleware applications, which make little or no assumptions about the semantics used to communicate between two entities, employ the concept of metadata in one form or another.

Metadata is data that describes other data. For example, a library catalog holds metadata about the books in a library (data), or a file system holds permission information about files. This metadata can be queried to give information about the structure and meaning of the data. This is a powerful tool because it allows applications to discover services at runtime and adjust as required. An application querying a database, for example, can discover the structure of the data and then use that information to query the data. This lends itself well for loose coupling, because an application only needs to have limited knowledge about the structure of the database at compile time.

Metadata has been incorporated into middleware and specifically event notification systems for different purposes [Sei99]. In content-based routing systems [RW97, CRW00, ASS+99, FLPS00] the data is routed according to the information contained within the message or event and it is therefore necessary for such systems to have metadata that describes the structure of the data. Systems that have incorporated storage into their architecture [BHM+00, Spi00] have needed metadata in order to allow for a uniform means of transmission and storage of events. Other systems have used metadata to describe event or message structures simply because it provides a more generic approach that can be used in a variety of application domains [MB98, BBHM96, BBMS98, BMB+00]. Below, we now describe the most important event systems which have made use of metadata within their architectures.

2.2.1 Message Oriented Middleware

Although not strictly an event-based system, message-oriented middleware deserves a mention in this section. It is the earliest commercial middleware product and is widely used and well established. MOM systems are also interesting because they use message type definitions to identify messages. They were originally used to reliably connect application programs with database servers, but have in the meantime also found a more general application, providing reliable communication between distributed applications.

MOM systems define a set of message types and asynchronous communication channels. A channel has associated with it a set of message types which it handles. Message *senders* submit messages to a channel. This channel is either connected to a receiving application or to an intermediary that duplicates or filters the messages. Sender and receiver do not need to be aware of each other. MOM systems are therefore decoupled both in space and time as the receiving application does not need to be running while messages are placed on the queue. MOM systems provide transactional guarantees when placing or retrieving a message from a queue and if sender and receiver are not in the same protection domain while delivering the message from one endpoint to the other. Some MOM systems provide a type system beyond the simple message type definition, but this is not always the case.

Many implementations of message oriented middleware exist. The most notable ones are IBM's *MQSeries* [IBM99a, IBM99b], Microsoft's *MSMQ* [Cor99] and Tibco's *Tib* family of products [Inc99].

MQSeries defines messages in terms of named queues, which can be connected into a directed graph for classification. The name serves as the topic of the messages in the queue. *MQSeries* queues can transmit messages of all types, unlike many other MOMs. Two operations are supported for the client to retrieve messages from the queue. These operations can be both blocking and non-blocking. The first operation, *take first* retrieves the first message from a queue. The second, *take cursor*, allows the client to select a group of messages from the queue. Selection is based on *message identity*, *correlation identity* or *group identity*. These three identities are defined by fields in the header of a message.

Messages are passed within a transaction. *MQSeries* thus offers a feasible solution to business applications such as a bank audit system, where it is of primary importance that messages are not lost. A message may be encoded in certain formats agreed by the publisher and the subscriber. A formatter needs to be used for different applications to reformat shared messages. For instance, a message containing four fields such as *first name*, *surname*, *account number* and *transaction type* in one application may be reformatted into a three-field message such as *name*, *account number* and *transaction type*. Other rules may be defined for transformation based on the format and content. Both format definitions and rule definitions are stored in a repository.

IBM's *Message Broker* builds on top of *MQSeries* to provide *publish/subscribe* functionality. Subscriptions are topic based, but structure can be implied through the use of deliminators. For example, *"Mail/*"* matches all published mail messages, whereas *"Mail/Peter"* only matches published mail messages from Peter.

2.2.2 CORBA Events

The Object Management Group has defined two standard event services, the CORBA Event Service [OMG01] and the CORBA Notification Services [NEC98]. The CORBA Event Service introduces the concept of an event channel with suppliers and consumers of events. An event channel is an intermediate object which decouples the supplier and the consumer. Event communication can be untyped, in which case a single parameter of type `any` is used for passing events. Applications can cast any type of data into this parameter. For typed event communication, an interface `I` is defined in OMG *IDL*. In the typed push model, suppliers invoke operations at the consumers using the mutually agreed interface `I`. In the typed pull model, consumers invoke operations at suppliers, requesting events, using the mutually agreed interface `Pull<I>`. The main purpose of the metadata is not for defining event types, but rather for defining the interfaces used for communication between event suppliers and event consumers.

The CORBA Notification Service extends the CORBA Event Service with event filtering, support for Quality of Service (QoS) and provisions for structured events. Structured events have a header and a body. The header consists of a fixed and variable part. The fixed part has three attributes of type string, `domain_name`, `type_name` and `event_name`. The first describes a domain name in which the event is defined. The second attribute, `type_name`, uniquely identifies an event within a domain. The last gives the specific event type a name. The variable header part contains information for QoS constraints. These are `EventReliability`, `Priority`, `StartTime`, `StopTime`, and `Timeout`. The body part of the structured event allows for the definition of typed and untyped attributes. The types allowed are the normal CORBA types.

The CORBA Event Notification service mentions in its specification an Event Type Repository. This service is optional. It allows suppliers to publish their event types and consumers to query it for event discovery.

2.2.3 Java Events

Jini [AAAF01, Mic01] events were introduced into *Java* through the Abstract Window Toolkit (*AWT*) and JavaBean releases [Jav97]. JavaBean events support a *register/notify* paradigm on a single Java Virtual Machine (JVM). An event object, derived from the class *EventObject*, can be sent to objects implementing the *EventListener* interface. When an application begins to span multiple VMs, and components are created on the fly without notification to other components, the *AWT* model of tight coupling between consumers and producers of events becomes very difficult.

Jini was not designed for widely distributed implementations. It uses a centralized lookup service and a multicast protocol. The more recently defined Java Message Service (*JMS*) [HBS$^+$02, Rou02] is potentially more scalable. *JMS* specifies an API for applications to use message-oriented middleware and supports both the message queue and *publish/subscribe* model of communication. An object is created as the destination through which messages are sent/received or published/subscribed by the *JMS* clients. This is similar to the indirect event communication model where a mediator is used for receiving and dispatching events. *JMS* does not support direct source to sink event communication.

JMS message have a message header and a message body. The header contains various fields, including specifications for destination, delivery mode, expiration, priority, message identifier and timestamp attributes. The header also contains a type field. *JMS* does not require a type repository, but the specification states that a repository can be used to describe the message types. The header can also be extended to include application specific properties. These properties can be of the basic *Java* type and are set by the message source.

The body of a *JMS* message can have different formats. These are:

Stream Message The message body contains a stream of Java primitive values and must be filled and read sequentially.

Map Message The message body contains a set of name/value pairs where the name is of type string and the value can be any of the *Java* primitive types. Values in this message type can be accessed sequentially or randomly.

Text Message The message is of type `java.lang.String`. This was included to support messages in XML format.

Object Message The message is a serialized *Java* object.

Byte Message The message is a stream of uninterpreted bytes. This is used for encoding a message to match an existing format.

JMS provides facilities to publish message types available within the system. Message

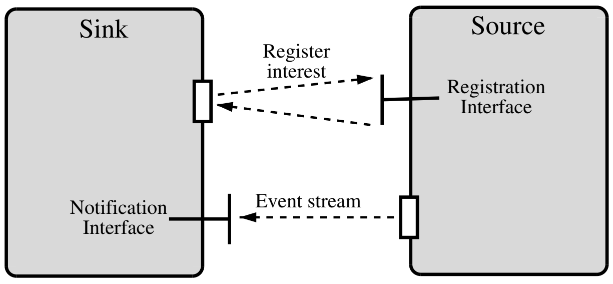


Fig. 2.2: The Cambridge Event Paradigm

producers publish their topics with these services while consumers query them for message topics and formats.

2.2.4 The Cambridge Event Architecture (CEA)

CEA [BMB$^+$00] in a framework designed and developed at the Computer Laboratory of Cambridge University. It was the first framework to use the *publish/register/notify* paradigm. CEA has been implemented on standard middleware platforms such as CORBA (see section 2.2.7), early RPC-based mechanisms and Java *RMI*. It has been used in building large, scalable, distributed systems such as network management and location-oriented applications [BBMS98].

In CEA, an event is a typed occurrence that is used for notification in the system. Each event is an instance of an event type. A service that is the source of events publishes, via middleware services, the event types it is prepared to notify. For scalability, a client must register interest in an event type by directly or indirectly invoking a register method at the source, with an appropriate filter expression. The filter expression defines a constraint on the events the client will be notified of. All registrations are recorded at the service and subsequently, whenever an event occurs at that source, the clients whose registrations match the occurrence are notified (see figure 2.2).

Besides this direct communication method, the CEA framework also makes provisions for indirect communication via mediators. These mediators can be used for various purposes. One of the main applications of a mediator is that it removes the filtering function from primitive event sources. Another area where mediators might be used is with mobile users or clients which may be disconnected temporarily from an event service. In these instances mediators can buffer event notifications until clients reconnect. CEA implementations have used mediators to implement a composite event engine. Here mediators, or composite event servers, handle complex event registrations. The registration of interest takes place with the mediator, which in turn registers interest with the services that produce the individual events making-up the complex event registration.

In [BBHM96], the CEA event model was enhanced to include event hierarchies and inheritance. This allows applications to register interest in a supertype, rather than having to register many times with individual services. Once again, these types of registrations are handled by mediator applications.

Although later implementations of the CEA framework were able to provide support for loose coupling between components [Spi00, PB02], the original CEA did not support loose system integration. Event clients wanting to interact with event services, and vice versa, need to know about each other's interfaces at compile time. Nonetheless, CEA was one of the first systems to introduce the concept of event typing, as well as the notion of mediator service to handle more complex interaction between entities within an event system. Finally, the enhancement of the event model to include event hierarchies and inheritance was an important step forward.

2.2.5 The Extended Cambridge Event Architecture (ECEA)

ECEA [BHM$^+$00] is based on CEA but has extended the *publish/register/notify* paradigm to provide a unified approach for event storage and transmission. In ECEA, events are defined in the Object Definition Language (ODL) of the Object Data Management Group (ODMG) [CBB$^+$99]. Every event is derived from a base type called *BaseEvent*, which cannot be instantiated. The *BaseEvent* type holds the header information of each event occurrence. It contains a unique ID (consisting of a unique object ID and a source definition for the object), a priority tag and a timestamp. Complex event hierarchies can be constructed through the use of simple inheritance (see figure 2.3).

Event definitions are compiled by an ODL parser and stub generator, which translates the ODL files into stubs generated as wrappers around event types. The stubs present the application programmer with an intuitive programming interface. *EventSource* and *EventSink* classes allow for a simple connection to the library and service object creation.

In ECEA, an event schema is maintained for each event type. As the ODL files are compiled, the metadata is stored in an event repository. The repository is ODMG-compliant, although this is not a requirement. If it is compliant, the ODL compiler of the store is used and the event metadata is stored in the repository. Mediators are used

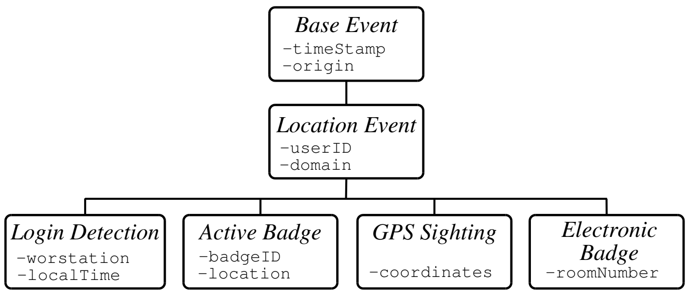


Fig. 2.3: Event Inheritance

to translate from the ODL representation to the internal representation of the store if it is not ODMG compliant. Applications can query the repository for dynamic event discovery.

2.2.6 HERALD

HERALD [Spi00] is a generic storage-enabled event architecture with many of the CEA features. It uses the ODMG'S type system [CBB$^+$99], and associates an *event schema* with every event type. It allows for event hierarchies and inheritance. Event consumers do not need to have prior knowledge of event producers. This allows for loose coupling and a dynamic system.

In HERALD, all events are derived from the base event type *event.* This is an abstract type, which holds the necessary header information. This includes a timestamp attribute, an event source identifier and attributes for security purposes. Furthermore, events can be tagged as *live*, *past* or *replay.* These values indicate whether an event comes from a store or whether the delivery of an event has been delayed. Every event type is described by an event schema. Besides describing the structure of the different event types, each schema is tagged with a version number to support schema evolution. Event schemas are stored with an event broker where details can be obtained by clients.

HERALD was one of the first systems to introduce event evolution. This provided support for the often dynamic environment of distributed systems. HERALD also provided a simple yet powerful generalization of the event storage concept. It viewed event stores as event consumers registering interest with mediators or directly with event producers. The events received by the storage nodes are immediately placed in the event store. A general purpose store can be used, thus making use of its powerful querying functionality and transaction processing capabilities.

Besides supporting dedicated storage nodes, HERALD also provides the capability to embed storage functionality within core modules, such as event producers, event consumers and mediators. Embedding event storage within event producers and mediators is useful for fault-tolerance and error-handling. The concept here is similar to message queues. Embedded storage in mediators is useful for supporting disconnected clients. HERALD exports interfaces which let applications decide for how long and for which event types histories are to be kept.

2.2.7 COBEA

In COBEA [MB98], a CORBA-based CEA implementation, a typed event is implemented with a pair of interfaces, the typed source interface supporting a register operation and the typed sink interface supporting a notify operation. The sink and source objects have an IOR (Interoperable Object Reference) and are registered with the ORB. A client calls the operations using the IOR through the ORB.

The scale of an application, with a large number of different event types, can make this architecture unmanageable, since every event type has one or more CORBA objects, which have to be registered with an ORB. COBEA therefore focuses on fully-fledged CORBA objects only for source and sink objects. A pair of generic interfaces (*Src* and *Snk*) are defined, with standard register and notify operations for all event types. The interfaces are implemented through a generic library. With this generic library approach, event type checking is carried out at run-time. The event type name (which identifies the type) and all of its attributes must be supplied at the time of registration. Wildcard parameters are supported. Event occurrences are matched against the registrations at the source/mediator and if the type name as well as the types and values of the parameters match, clients are notified accordingly.

In COBEA, event objects are defined using OMG's Interface Definition Language (*IDL*) [OMG00]. Applications wanting to interface with the event service incorporate the pre-compiled stubs at compilation time. Event sources use an extended version of *IDL* to publish the event types they produce and can notify. Event stubs are generated automatically and linked with the client code. These objects have a register method, allowing clients to register with the services.

The COBEA implementation showed that CEA could be built on top of a standard middleware platform. It made use of the extra services of that middleware and demonstrated that asynchronous operations and more complex event services could be added to inherently synchronous platforms.

2.2.8 READY

READY [GKP99, GKP00] is an event notification service built on top of the *omniORB* CORBA ORB [LP98]. It uses both the CORBA Notification Service structured events and grammar for filter expressions. Clients interact with the READY system using *admin*, *supplier* and *consumer* sessions. Admin sessions are used for administrative operations and creating/destroying supplier and consumer sessions. Supplier sessions are used to supply events to the service, while consumer sessions are used for registering specifications. Specifications are expressions which specify a pattern and an action to take when the pattern occurs. A typical action is *notify*, which sends a notification to the subscriber if the filter expression is matched by an event. This notification is sent via a callback function. A specification can be suspended and resumed using a consumer session.

The general form for a simple matching expression that matches a single event is:

$$event\_var : event\_type \mid expression$$

A READY expression is any legal expression from the filter grammar specified in the OMG Notification Service [Obj00]. Simple expressions can be combined to form compound matching expressions. The operators supported are `"&&"` (and), "||" (or), `";"` (then) and `"not"`. Another from of a compound matching is to specify a sequence of events which all match the same expression. The general format for this is:

$$event\_var[j...k] : event\_type \mid expression$$

The variables j and k indicate the minimum and maximum number of events in the sequence to which the expression is applied.

READY clients can be partitioned into event zones or domains. These zones are determined by administrative or logical boundaries. READY has boundary routers which connect the different zones. Boundary routers support event translation using mapping functions. These mappings translate between different READY event types. This provides support for cross domain event notification of different READY applications.

2.3 Internet-scale Event Notification

Newer event architectures have focused on scalability issues. Their aim is to provide an architecture that is scalable, having the capability to connect heterogeneous components on an internet scale. Events are routed via event brokers, distributed across the network, according to the content of the event instances. This content-based routing approach allows to distribute event filtering across a network of brokers so that a larger number of event sinks and sources can be supported than in a centralized approach. Global naming becomes redundant because the name-to-location binding between event types and event sources is performed by the network of brokers.

The primary aim of wide-area notification services is to be scalable. In these systems, the difficulty always lies in maximizing expressiveness without sacrificing scalability. Expressiveness refers to the selection mechanism of events, whereas scalability refers to the delivery mechanism of the system. The more one can express in the event filtering language, the less scalable the system becomes as filtering complexity increases. In these systems, messages are routed from the event source to the client via brokers. If the event registration mechanism is simple, routing event instances is less difficult. But with increased complexity, the routing becomes more expensive, meaning brokers take proportionally longer to determine how to correctly route a message. A system that only supports type subscriptions requires the brokers to know which clients have subscribed, whereas a system that supports attribute-based subscriptions also requires the brokers to match event instances against subscriptions before routing them.

Routing packets within a network is normally straight forward. The packet contains a destination address, which a network node retrieves and uses to route the packet accordingly. Within a content-based event notification system, this becomes more difficult. An event service may have millions of subscribers. When an event occurs which multiple subscribers are interested in, the event needs to be routed to all of these subscribers, which may be highly distributed across the network. The event itself does not contain information that identifies all the subscribers. In a content-based messaging or event notification system the nodes or brokers have the necessary information to route events or messages correctly. In this section we describe three specific content-based messaging systems.

2.3.1 Elvin

Elvin [FMK$^+$99, SAB$^+$00] is a general purpose event notification service. It uses a client/server architecture for delivering notifications. Clients establish sessions with an Elvin server process and then send notifications for delivery or register to receive notifications sent by others. Clients can act as both producers and consumers of information within the same session. Elvin supports content-based routing, delivering unaddressed notifications based on the information they contain rather than the direction they are

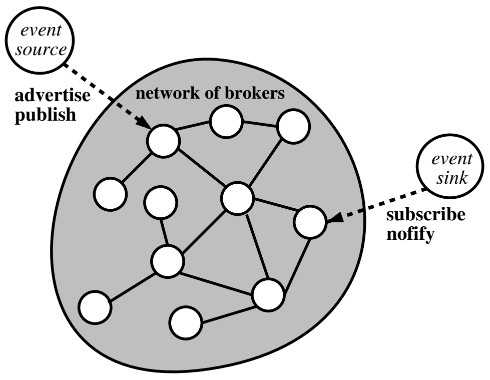


Fig. 2.4: A Content-based Routing Architecture

sent. A data source does not need to be configured with any information regarding recipients. The architecture supports source-side filtering using a mechanism called *quenching.*

Messages sent through Elvin are simple, yet extensible. An Elvin notification is an arbitrary length list of name/value elements. Each element has a simple name and can take a value from the following basic types: integers, floats, strings or an array of bytes. Below is an example of an Elvin event instance:

```
sensor:    "thermostat 123"
reading:   34.24
time:      234185638
```

Since Elvin does not use any part of the notification to specify delivery, applications are free to send and able to receive notifications with fields that are not necessarily used by the clients. This flexibility allows applications built using Elvin to evolve.

The subscription language used in Elvin provides matching functionality like many programming languages use for conditional tests. Subscriptions are composed from logical expressions across the values of a notification using a simple C-style syntax. Below is an example of a subscription:

```
sensor == "thermostat 123" && reading > 30.0
```

Elvin is currently targeting local-area clustering of servers to provide resilience in the case of individual server failure. These clustered servers cooperate to share the load of subscription evaluation, client connection handling, network bandwidth and CPU usage. Future work aims to focus more on the issues of wide-area federation of Elvin clusters. The aim of Elvin is to support a global-scale network of linked Elvin domains, with message routing across the internet.

2.3.2 Siena

Siena [RW97, CRW00] is a content-based notification system which aims to balance expressiveness and scalability. Underlying Siena is a data model that drives the semantics of the service. Notifications (event instances) are untyped with typed attributes. Attributes have a *type*, *name* and *value*. The notification as a whole has a structural value derived from its attributes. For example, a notification may have the following format:

```
string  class    =  location/position
time    date     =  Mar 21 10:59:08 MST 2001
string  busID    =  1234
string  bus_stop =  Chase Street
```

The types supported belong to a predefined set of primitive types generally found in programming languages. These types have a fixed set of operators defined for them. Siena does not support typed notifications because this would imply a global authority for managing the type space. On an internet-scale this is not feasible. Using a predefined set of types to construct notifications allows the notification service to optimize the delivery of notifications.

2.3.3 Gryphon

Gryphon is both a commercial product as well as a research project. Both are developed and maintained separately. The commercial product has been widely used, specifically in sporting events for result and score dissemination. The Gryphon described here is the one being developed as part of the research project.

Gryphon [ASS+99, BKS+99, SBC+98] uses subject-based *publish/subscribe* systems as a starting point and has extended this model. It has added the concepts of content-based *publish/subscribe*, stateless event transformation and event stream interpretation. None of these extensions by themselves are novel, but the combination of them makes Gryphon an interesting event notification system.

Gryphon associates a schema with each event stream, thus allowing for a content based *publish/subscribe* mechanism. Its type system is more complex than that of Siena. It supports simple types as well as collection types. The simple types are atoms (uninterpreted data), strings and integers. The collection types supported are bags, lists and tuples. Events themselves are typed, not just the attributes. Clients can subscribe to events with a specific set of attributes rather than just a specific event type.

The stateless event transformation provides a mechanism to group similar but different events together. Events can be transformed into new types according to transformation rules. Event stream interpretation gives the system the ability to interpret streams of events rather than just individual events. Clients can register interest in an event which is made-up of a combination of events.

Both Siena and Gryphon have shown the importance of event schemas in content-based notification systems. Although they take different approaches and focus on different aspects of the event notification architecture, their contributions are very important and they have shown that internet-scale notification is feasible.

2.4 Summary

This chapter has outlined the development of middleware applications over the past two decades. It began by describing the very early systems trying to provide a uniform interface for applications to connect to database systems or other information sources. It has shown how asynchronous communication found in *publish/subscribe* architecture has decoupled the entities within a system and thereby provided a mechanism for loose rather than tight coupling. As systems became more complex the need for genericity and scalability became important. These issues were addressed by introducing metadata into middleware, specifically messaging and event notification architectures. We showed how typing become essential for internet-scale event notification systems and architectures which provide event storage.

The concept of metadata has not only grown in importance for existing middleware architectures, it is also essential for developing the event federation paradigm described in chapter 4. Before describing this paradigm, we look at federations and how federations of autonomous systems are constructed, especially database federations.

3. FEDERATION

We encounter the concept of federation and the problems and difficulties that accompany a federation almost everyday. Nowhere is this more the case than in politics. Over 200 years ago the United States of America was founded as a federation of autonomous states. Only recently Europe has begun the process of creating a federation of European states linked by common interests, markets and a common currency. Whereas the US is a federation, Europe is in the process of forming one. Europe is dealing with the issues that confront entities, in this case countries, when creating a federation. Countries wishing to join a federation must constantly evaluate the benefits of a federation against the cost of giving up part of their autonomy. A key characteristic of any federation is therefore the style of cooperation among otherwise independent components. Even though the components become part of a larger entity, the federation, individual components continue to function on a local level.

These ideas are very similar in computing. Here, two very different approaches exist to federating systems. The first involves mapping data from independent systems to a global system. The second approach involves mapping directly between different systems. This concept can be illustrated by considering two different databases. Using the first approach one conceptually creates a new database into which each of the two databases would map their data. The second approach requires components to translate data directly from one system to another and vice versa. In the first approach, users wishing to get data from the federation have to access the federated system, whereas in the second approach users only access the local database, but receive information from both data sources. Both approaches share common concepts, but each approach is subtly different, especially with respect to the coupling model.

In the previous chapter we encountered the concept of loose and tight coupling between middleware components. There exists a similar notion for federations. The concepts are slightly different but related. In a distributed environment we referred to coupling when discussing how much prior knowledge entities have of each other before cooperating and how closely they are linked once they cooperate. In a federation we speak of the degree of coupling when referring to the autonomy a system has within a federation. To come back to our political example, we say that European states have very little autonomy if the European parliament controls or influences most of the activities of its states. They are therefore tightly coupled. If the members make most of their own decisions on a local level without being influenced by the federation, we say that the states have a high degree of autonomy and are loosely coupled. This is an important concept and is discussed in more detail in this chapter and the next.

In the remainder of this chapter we discuss federations by looking at how they are used in areas of computer science. The best-known area is databases, where research into database federation has been conducted for more than two decades. In section 3.1 we describe and evaluate the standard reference model for database federation developed by Sheth and Larson [SL90]. Following that, in section 3.2 we look at mediator technology in database federations. Section 3.3 describes the problem of semantic heterogeneity that any federation is faced with and discuss some approaches to overcoming this problem.

3.1 Database Federation

Database federations connect heterogeneous databases, making them appear as a single entity, while still allowing component database to retain a degree of autonomy. It is generally important that new databases can join existing federations relatively easily and without much programming effort. This is more easily said than done. Much research has been undertaken in this field and although vast improvements have been made in the last two decades, data integration problems still exist and probably will for a long time. In the beginning of this section we show why this is the case. We follow this with a look at the reference model [SL90], which has been instrumental in understanding the problem of federating databases.

Federating databases can be divided into two separate, but related disciplines. These disciplines have to do with the types of heterogeneity the federation has to resolve in order for the databases to be able to cooperate. The first entails resolving system and application heterogeneity. This means getting a machine of type A running database X on a particular operating system (OS) to communicate with a machine of type B running database Y on yet another OS. Here, communication refers to the concept of linking applications in a simple and reliable manner. To a certain degree these issues have been resolved. Database management systems such as *DB2* and *Oracle* run on multiple platforms. Furthermore, standards such as *ODBC* [Net] and *JDBC* [FCHH99] have helped eliminate or reduce the problems of heterogeneous hardware and system software. These standards specify an API for connecting to remote database applications in a uniform manner. Using these standards, an application can connect to multiple databases.

With enough knowledge the application can integrate the information from the different data sources. It can then be made to appear as a single database to other applications. This is very hypothetical because many different issues are involved that we are neglecting here for the purpose of simplicity. The knowledge mentioned above that this application requires to be able to merge the different data sources is knowledge about the semantics of a data source. Resolving semantic heterogeneity is the second discipline involved in database federation and heterogeneous information integration [BB99] in general. Resolving semantic differences means trying to reconcile and understand what different databases mean when they describe the data they maintain. The problem begins by looking at the different schemas of databases and reconciling these. The task of schema integration is discussed in this section and the problem of semantic heterogeneity in section 3.3.

3.1.1 A Reference Architecture

Classical database federation is described in terms of a reference architecture [SL90]. This reference architecture helps to clarify various issues and provides a framework to help understand, evaluate and compare different architectural options involved in federating databases. Furthermore, having a reference architecture clearly separates the different components in federated systems. With this, those components of interest for event federation can be highlighted without bringing irrelevant components into the discussion.

[SL90] describes the different components of the federated database reference architecture as follows:

Data The facts and information managed by the database

Database The repository of data structured according to a data model

Commands Requests for specific actions

Processors Software modules that manipulate commands and data

Schemas Descriptions of the data managed by the database management system

Mappings Functions that transform or correlate schema objects from one schema to schema objects of another schema

[SL90] argues that a combination of these components can describe any centralized, distributed and federated database system. Furthermore these components hide many of the implementation details that are not relevant to understanding different architectures. For the purpose of this dissertation we focus only on two components, schemas and processors. They are the most important components for a database federation, but also for eventually describing event federations.

Schema Components

[SL90] propose a five-level schema architecture for federated databases. This architecture is based on the ANSI/SPARC three-level schema architecture [Ins75, TK78] for centralized databases. We begin by describing this three-level schema architecture. The three levels of this data description model are the conceptual schema, the internal schema and the export schema.

The Three-level Schema Architecture The *conceptual schema* describes the logical and conceptual data within the database and their relationship. It tries to describe all the data that may be of interest to the users of the database. This data is generally expressed in terms of the internal data definition language of the database. For example, in case of an ODMG compliant database this would be the description of the data given in *ODL*.

The *internal schema* of a database describes how the conceptual data is stored within the database, in other words the physical characteristics of the conceptual data. This may include such information as the physical location of tables on a disk or index information for records within a table. The separation of the conceptual and internal schema allows for data independence in the ANSI/SPARC model. For example, if a new index is created for a table, the conceptual data is not affected and applications accessing the database do not need to adjust to this change.

The *external schema* describes the subset of data that can be accessed by specific users. Not every user needs to see all the data within the database. There may be an external schema for each individual user of a database.

The Five-level Schema Architecture The three-level schema architecture is inadequate for describing architectures of federated databases. It must be extended to support the dimensions of a federation, namely heterogeneity, distribution and autonomy. The description here is based on [SL90], but others have developed similar concepts [HM85, LA86, Bla87]. The five-level schema architecture consists of the local schema, the component schema, the export schema, the federated schema and the external schema.

The *local schema* is the same as the conceptual schema in the three-level schema architecture. It is a local schema expressed in the local data model. Each database in the federation may use a different data model to describe its data.

The *component schema* is the local schema expressed in a common data model. The common data model depends on the federation, but it is important that a data model is chosen so that as little information as possible is lost when translating from the local schemas to the component schema and vice versa. The purpose of this schema is twofold. Firstly it represents all the data within the federation using a single data model. This resolves heterogeneities caused by different data models. Secondly, information or semantics missing from the local schema can be added to the component schema. The translation from the local schema to the component schema serves as the basis for the mappings from local to component objects.

The *export schema* in the five-level schema architecture is very similar to the external schema of the three-level schema architecture. It is the schema that describes the data that the component database is willing to share with the federation. It is a subset of the component schema. It may include access information describing the rights of specific federation users.

The *federation schema* is an integration of the export schemas of a federated database. This schema handles the distribution aspect of a federation, meaning it contains the information which details where certain data resides within the federation. A processor component handles the transformation of commands from the federated schema to the export schema.

The *external schema* defines the schema for a set of users or applications. It allows for the addition of access control information and additional integrity constraints. Since federated schemas can be very complex, the external schema can contain a subset of information that might be interesting to an application or a set of users.

[SL90, Cat91, SCGS91] have argued about the importance of the data model used for representing schemas on a federation level. As mentioned, the data model should be able to represent all the information contained within the component schemas. Extra expressiveness can be helpful in resolving semantic heterogeneity issues.

Processor Components

As mentioned, processors are components that manipulate data and commands. Both of these aspects are important for event federation and are not only important for federated databases, but also for centralized ones. [SL90] define four types of processors, transforming processors, filtering processors, constructing processors and accessing processors.

Transforming processors transform commands or data from one language of format to another. This provides a form of data model transparency. Data model transparency hides both the difference in query language and data format between databases. For example, using the mapping components between two schemas, a transforming processor transforms data from a table format of a generic relational database to an object-oriented format. Similarly, it transforms an *OQL* query into an *SQL* query. Consider a federated architecture where the component database is a generic relational database and the federation uses an object oriented data model such as the one described by ODMG [CBB+99]. Here both command and data transformations occur. When a query is sent to the federation in the command format of the federation, the federation needs to translate from the global to the component format. Similarly, when the data comes from a component database to the federation, the data must be transformed to the federation data format. Mappings between different components can either be encoded in the logic of the transforming processor or they can be stored separately and accessed by the transforming component when required. The latter is more generic and preferable.

Filtering processors limit or constrain the data that can be passed between processors or components. Again, each of these filters is associated with a mapping which describes the constraints on the data or commands. Filtering processors check the syntax of commands before passing them on, perform semantic integrity constraint checks and verify who may perform commands on which data. Data that is passed from the component database to the federation, for example, may contain multiple data entries with the same key value in the federated schema. The filtering processor checks these constraints and gets rid of the duplicate key values.

Constructing processors partition or replicate an operation submitted by a single processor into an operation accepted by multiple processors. Similarly, they merge data received from multiple processors into a single data set for the use by a single processor or component. Thus, these processor provide distribution and location transparency because other processors do not need to know the details about data partitioning or

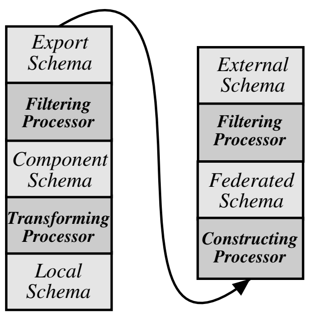


Fig. 3.1: Schema and Processor Interaction

data distribution. For a query submitted to the federated database, the constructing processor partitions the operations and submits the correct queries to each of the component databases. When receiving the result from each of the component queries, it merges the data and returns the result as a single data set. Besides partitioning queries and merging data, the constructing processor is also responsible for integrating different schemas into a single schema, negotiating the content of the global schema and global transaction management.

Accessing processors take commands from several processors and manage the execution of these. The major issues addressed by this processor are concurrency control, backup and recovery.

Processors and Schemas

As described, schemas, mappings and processing components interact with each other and are central to the functioning of a federated system. The schema components provide data transparency necessary to deal with the heterogeneity of the different data models of the component databases. The processor components provide distribution and location transparency. When describing the event federation paradigm in chapter 4 we shall see these components again. Having a clear understanding of each of these components and their functionality is helpful when constructing similar modules for event federations.

3.1.2 Loosely and Tightly Coupled Systems

Whether a database federation is loosely or tightly coupled is determined by the schema integration process, specifically the creation of the federated schema in the five-level schema architecture. In a loosely coupled approach, the process is driven by federation users. [HM85] refers to this as schema importation. Typically, users create a federated schema by looking and evaluating the available export schemas. A tool is generally used to create the federated schema, providing mappings from the export schemas to the federated schema. Here the user is responsible for understanding the semantics of the export schemas and resolving semantic heterogeneity. Once created, the federated schema belongs to the user and creator and can be deleted or updated any time without having to consult a higher authority. Important in this approach is that each of the component databases has created export schemas without having consulted each other or a federation administrator.

In a tightly coupled federation the scenario is different. Here a federation administrator creates a federation schema. The export schemas of the individual component databases are created in negotiation with the administrator. The administrator usually has access to the component schemas to help determine which data should be exported. Solely the administrator is responsible for managing the federated schema.

These two approaches may also be combined, whereby multiple component database administrators may be involved in the process of creating a federated schema and individual export schemas. Here we have only shown the two extremes. A situation where all the administrators of the component databases form the federation authority may represent such a hybrid approach.

A federation can be either *static* or *dynamic*. A tightly coupled federation tends to be static since the federated schema evolves very slowly and tends to not change very much. A loosely coupled system tends to be more dynamic since federated schemas can be created, deleted or updated at any time.

There are obvious trade-offs between the two approaches. A tightly coupled system is desirable in an environment where system control is essential. This is usually the case in traditional business or corporate environments. In such an environment users would find it too difficult to perform negotiation and integration themselves. A central authority with complete knowledge about each of the individual component databases is important and often invaluable. A tightly coupled system is also necessary when update operations are possible in the federated system. This has to do with semantic differences between the components which individual users are possibly not aware of. The requirement for location, distribution and replication transparency is also a strong argument for tightly coupled systems.

A loosely coupled architecture has advantages in an environment where a large number of autonomous, read-only systems are to be linked [LA86]. Since the systems are read-only, updates are not a problem. This type of system also allows individual users to determine the semantics of other export schemas in a way that is more suitable for their needs. Users can determine mappings as they want and not as the federation administrator dictates. Consider an export schema with an attribute called *category* in a *cars* table. The domain of values for this attribute may be *truck*, *sedan*, *sports car* in the federated schema. Different users may want to map a sports utility vehicle from another database differently. In the loosely coupled approach a user would have this flexibility.

It is important to remember that, as [SL90] states, the user of a loosely coupled federation has to have knowledge that enables him/her to find the appropriate data in export schemas and to define mappings between the federated schema and export schema. Lack of adequate semantics in the export schemas makes this very difficult. Once again, the semantics supported by the data model of the federation is essential in a loosely coupled system. Loose versus tight coupling will be an important design aspect in the event federation paradigm in chapter 4.

3.2 Mediators

The reference architecture described in the previous section has focused on federated database systems. A more generic but similar problem involves integrating heterogeneous information or data sources, not necessarily databases, via mediators. [Hul97] describes mediators as integrated read-only views of data that reside on multiple databases. Although most research in the area of mediators focuses on integrating data from database management systems, we argue that the methods of integration described in mediation research are relevant to any form of data source which is described by metadata.

Initially, mediation was mainly concerned with the problem of information overflow and its solution. This has changed and mediation is now closely related to the idea of information transformation or translation. We look at general approaches of integrating different information sources. The concepts are interesting because they propose solutions not just for databases, but for information sources in general. Furthermore, the solutions are of a loosely coupled, rather than a tightly coupled nature.

The model of mediation between entities such as computer systems or applications was developed by Wiederhold [Wie92, WG97]. In his model Wiederhold distinguishes between *data* and *knowledge*. Data describes specific instances and events, whereas knowledge describes abstract classes. Knowledge needs to be gathered and formalized and can be verified or disproved by data. Based on these definitions, Wiederhold defines a mediator as a software module that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications.

Mediation is thus not an entirely new concept. In the previous section, we were introduced to processors which transformed data according to some knowledge. Mediators focus on the process of transforming data to make it useful and meaningful to the application using the mediation service. To do this, the mediator requires some knowledge to perform the transformation. In the previous section this knowledge came in the form of mapping components and logic which applied the mappings to data. We evaluate a few of the mediation services available and evaluate the major components within these mediation services. These components also play an important role in the event federation paradigm discussed in the next chapter.

3.2.1 The Partial Schema Unification Approach

[HM93] proposes a partial schema unification approach for integrating heterogeneous information sources. Unlike in the five-level schema architecture, in this approach multiple import schemas [HM85] or federated schemas [SL90] can exist. In a partial unification approach systems no longer need to integrate their data into a single global schema, which is often difficult and costly. The focus in [HM93] is on partially uniting metadata specifications. Metadata refers to the data that represents the structural part of a database (the conceptual schema) as opposed to the actual data. The key in doing this once again lies in resolving semantic heterogeneity. This means determining the relationship between objects that model similar information and determining possible conflicts in their representation.

The task of interoperability is divided into three subtasks, which are performed iteratively during information sharing in a federation. It is important that these tasks are interactive because in most instances human input is essential. The first phase is the *resource discovery and identification phase* and involves locating all sharable information within the federation. This is stored in a semantic dictionary, which can be accessed at any time by any member of the federation. The discovery process is done by an *intelligent sharing adviser* querying the export schemas of the federation, which contain all the non-private objects. In the second phase, *resolution of semantic heterogeneity*, the relationships between sharable objects are determined. This phase uses meta-functions to discover structural information about the objects. The information tries to resolve semantic uncertainties. In the final phase, the *sharing and transmission* phase, the most efficient access to the shared objects is determined.

Information Discovery

The first and second phase of the mediation process described by [HM93] are particularly interesting. For a mediation service to gather information, the participants of the federation must express the information they are sharing in a common and semantically rich data model. As mentioned previously, this model must be able to express as much information as possible so as to capture the intended meaning of the shared information. Once the export schemas have been created, a component can discover the available resources. Here one can imagine a user wishing to discover information using a tool such as the *intelligent sharing adviser* to gather pertinent information. Once possible data sources have been discovered, the semantic heterogeneity must be resolved.

Tools for Resolving Semantic Heterogeneity

A well known mechanism for resolving semantic heterogeneity is testing for structural and behavioral equivalence. Structural equivalence is easily tested when dealing with simple, atomic types. It becomes more difficult with complex data structures or objects. [HM93] propose meta-functions which return metadata information about remote data. These functions can take the form of *ShowAllTypes()*, which returns all types of an export schema or *HasSubtypes()*, which returns all subtypes of a type, and so on. Each remote data source wishing to participate in the federation must have meta-functions.

A local lexicon is used to define every entity or object that is part of an export schema. Different approaches can be used for representing information about objects. [HM93] provide a simple approach, where the lexicon is a static collection of facts of the form:

<term> relationship <term>

The term on the right hand side describes the unknown object on the left via the relationship. The terms on the right are taken from a list that characterizes the commonalities of the federation. The relationships can be any that might be useful for describing the meaning of the unknown entities. Possibilities are *identical, equal, compatible, kind of, association, has*, and so on. This lexicon complements the information from the meta-functions, since it is usually not possible to discover the meaning of a term or object by simply looking at its structure.

The last tool is a semantic dictionary, which is maintained by the *sharing adviser* mentioned above. While local lexicons contain the semantic meaning of entities or objects, the dictionary holds information about the relationships between objects.

Unification of Information

Once semantic heterogeneities have been discovered and are understood, they must be resolved. This is a two-step process. the first involves conflict resolution. Here naming conflicts and type conflicts are addressed and resolved. In the final unification stage more complex issues are resolved. [HM93] give different examples, but this process cannot be generalized and largely depends on the information to be unified.

3.2.2 The Wrapper/Mediator Approach

More recently, research has been undertaken into using a wrapper/mediator approach [GMPQ$^+$97] to integrate heterogeneous data sources. It uses wrappers around the data sources, which translate queries from a common query language to the local query language and data from the local to a common format, dealing with platform heterogeneity. Wrappers [MFO90, GMPQ$^+$97, WAC$^+$93] represent an extra layer of abstraction, leaving the mediators to deal with the semantic integration problem.

TSIMMIS [GMPQ$^+$97] argues that a more flexible common data model is required for mediation than is commonly found in database management systems. This data model must have the following characteristics:

- It must support a rich collection of structures, including nested structures, as is found in type systems of typical programming languages.
- It needs to handle missing information or related information of a different structure in a suitable manner.
- It must provide *meta information* about the structures themselves and about the meanings of the terms used in the data.

Furthermore, the mediation system must support a common query language to allow new mediators to join an existing system and new sources to provide input to existing mediators.

These concepts have been encountered before. The idea of allowing new mediators to join an existing system is interesting. The key to this is a common query language. As mentioned, the wrappers are responsible for interpreting the query language. The wrappers in TSIMMIS are able to determine whether the data sources are capable of understanding queries. In certain instances they may not be, especially when data sources are simple file systems, for example. When queries have been successful, wrappers translate the data they receive from a query into the common data model of the system. This leaves the mediators to convert data from the common data model to that of the end user system and to integrate multiple data sources. In the previous architecture the distinction between wrapper and mediator was blurred. Making the distinction as TSIMMIS and other wrapper/mediator architectures do, and using a global data model between the wrapper and the application, is a useful approach for the federated event paradigm.

3.3 Heterogeneity

We have continually encountered the concept of semantic heterogeneity in this chapter. In this section we classify and describe it in more detail. Heterogeneity is a broad term used in many environments, including many different areas of computer science. We differentiate between two types of semantic heterogeneity. The first is fundamental, the second structural heterogeneity. In this section we discuss both, but the primary focus will be on structural semantic heterogeneity [Col97, Hul97, HM93].

3.3.1 Fundamental Semantic Heterogeneity

Fundamental semantic heterogeneity is a more general form of semantic heterogeneity which cannot, even in principle, be resolved unless the data sources change their structure. Garcia-Solaco [GSSC96] presents an extensive taxonomy of fundamental semantic heterogeneity. This form of heterogeneity cannot be resolved algorithmically.

An example of this type of heterogeneity is the instance identification problem [WM89] or entity identification problem [LSPR93]. This occurs when two different information sources store information on an identical object, but do not share enough common attributes to identify the object as the same. Another example occurs when two descriptions in different data sources have similar meaning, but are not quite the same. Furthermore, neither data source (and schema) contains enough information to resolve this problem. For example, the sales figures in one database may include sales tax, while similar figures in a different database do not. If neither data source contains information to clarify this discrepancy, the fundamental semantic heterogeneity cannot be resolved.

Tight coupling is not possible when fundamental semantic heterogeneity is encountered. In these instances systems can only be linked if one or both change their structure to resolve the fundamental heterogeneity. Loose coupling is possible, but usually at the cost of information loss.

3.3.2 Structural Semantic Heterogeneity

Structural semantic heterogeneity can be resolved. It occurs when the same information is represented in two separate databases in structurally different but formally equivalent ways. It is generally a consequence of independent creation, design and evolution of autonomous information systems. We give a classification of structural semantic heterogeneity and explain how it arises and how it can be resolved.

Structural semantic heterogeneity arises for different reasons [BLN86]. The first is caused by a possibly different perspective when modeling the information during the design phase. The second is caused by the rich set of modeling constructs available within data definition languages which allow designers to model the same idea in many different ways. Lastly, heterogeneity may arise because of incompatible design specifications.

[HM93] defines a spectrum of heterogeneity based on five levels of abstraction.

1. The metadata or data definition language is different. This is also known as the conceptual database model mismatch.
2. The metadata specification differs and as a result the conceptual schemas are different.
3. Objects may be represented in a different manner, which results in naming conflicts, for example.
4. Systems represent low level atomic values differently.
5. The data may be managed by different database management systems.

Heterogeneity of the first type is dealt with by wrapper applications described in the previous section. The fifth level of heterogeneity is what we have previously termed system heterogeneity and can also be overcome by wrappers. Types one through four are the types of structural semantic heterogeneity which are most often encountered during system integration. Examples of the four types of structural semantic heterogeneity are given below.

Domain Conflicts The same object or entity is described differently in different domains. For example, Peter is known as *pp202* in domain A and *peter* in domain B.

Naming Conflicts The same attribute has different labels. For example, attribute *name* versus attribute *lastname*.

Type Conflicts Different types are used to describe related entities. For example, a temperature reading may be of type integer in one system and of type float in another.

Structural Conflicts A different data organization or structure is used to represent the same concepts. For example, an address type is represented as a structure or as a single attribute of type string.

These heterogeneities can be overcome when there is enough information in the metadata to clarify the meaning of each of the objects within the data source. When this is not the case, resolving the conflicts described above becomes difficult, and sometimes impossible.

Domain conflicts can be overcome by using *semantic dictionaries*. These dictionaries contain mappings between domains. In the example above, the semantic dictionary would contain an entry that identifies *pp202* as *peter* in domain B and *peter* as *pp202* in domain A. In situations where objects cannot be mapped 1:1 across domains, a more complicated mapping mechanism must be used, usually involving a mapping function. This can result in information loss. An example of this are two attributes in different domains used to represent grades or marks. One system may be based out of ten, while the other is based out of one hundred. When combining these sources, information loss occurs.

Naming conflicts are generally easy to overcome. They require mapping functions which simply change the labels of the attributes. Spotting naming conflicts is often more difficult then resolving them. For example, the information available may not immediately make it clear that a naming conflict exists. Two attributes may have very different labels that are actually related. Similarly, a naming conflict can be mistaken for a structural conflict. An attribute *price* may or may not include sales tax, for example.

Type conversion can be more difficult, but can usually be overcome by applying conversion functions. Programming languages, for example, provide functions for converting strings to integers. Structural conflicts are more difficult to resolve. This usually involves decomposition or composition. In the example above, the address type represented in a structure would either need to be decomposed into a single attribute or the single attribute describing an address would have to be used to composed a structure.

Usually the structural semantic heterogeneities described above occur together. This means that there may be structural as well as type conflicts for related attributes from different data sources. In this case conflicts are resolved sequentially within systems.

3.3.3 Overcoming Semantic Heterogeneity

One of the main problems when trying to resolve semantic heterogeneity is the ambiguity of data. Just looking at data and its metadata is sometimes not enough to fully understand what something means. This causes most of the difficulties in data integration. But, completely removing ambiguity from data is an unrealistic goal. [SSR94] propose using *semantic values* to facilitate integration of heterogeneous data and help resolve many ambiguities in data.

A value is an instance of a type. The semantics of the value are determined only by its type. If the value *31.2* is of type *Celsius*, then this value can not be directly compared with something of type *Fahrenheit*. Types adequately define values as long as the values are one dimensional. For example, a monetary value *4* of type *dollar* may have an additional dimension such as a scaling factor. Thus *4 dollars* with a scaling factor of 1000 is actually *4000 dollars*. To represent this extra dimension one can create a new type such as *thousandDollars*. This type would then more accurately reflect the real meaning of the value *4*. Defining types such as *thousandDollars* is impractical.

A solution to this problem is to define a semantic value that is the association of the context with a simple value. This idea is similar to property lists used in *LISP*. An atom in *LISP* might have the properties *currency* and *scaleFactor*. Again, if the value of the atom is *3* and the *currency* property is dollar and the *scaleFactor* property is 1000, then the atom denotes 3000 dollars.

A context is defined as a set, where each element of the set is an assignment of a semantic value to a property. This is a recursive definition, meaning that the value of a property can have another context. Using this definition, simple values become equivalent to semantic values without a context. The example below shows the use of semantic values. The type of the value defined is *Sales* and may be a value used for calculating annual profits.

250(ScalingFactor = 1000, Currency = 'US dollars', Period = 'first quarter'(StartDate = 'January 1'))

The value 250 has three properties, *ScalingFactor*, *Currency*, and *Period*. The value of *Period* is also a semantic value with the property *StartDate*. The semantics of 250 can therefore be determined as sales of 250000 US dollars for the first quarter starting on January 1.

Making the context explicit is useful for comparing different values and converting between them. Conversion functions for simple semantic values like *4(Currency = 'US dollars')* are easy to implement. The function *cvtCurrency(4, 'US dollars', 'Euro')* returns a simple value. These conversion functions can be implemented in any programming language and may involve table lookup or consulting online, real-time data sources (for currency conversions for example).

Conversion functions can be *total*, *lossless* or *order preserving*. An example of a total conversion function is a function that is defined for all arguments. A function that converts a value from inches to centimeters, for example, is a total function. It can be applied to any argument. An example of a non-total conversion function is given below.

'USA' = cvtLocationGranularity('New York', 'city', 'country')

This function does not hold for all arguments because certain locations that might need to be converted are not cities.

A lossless function is one where a value can be converted from one property value to another directly or in a sequence of steps. A conversion function that converts length units is lossless because it is irrelevant whether the conversion is done in steps (meters → centimeters → inches) or directly (meters → inches). An examples of a function that is lossy is a text format conversion. The function shown below converts the value of formatted value of s into an unformatted value s_1.

$$s_1 = \text{cvtFormatType}(s, \text{'html, 'unformatted'})$$

Converting s_1 back, via the function below, does not guarantee that $s = s_2$, meaning s_2 probably does not have the same value as s. The function *cvtFormatType* is therefore not lossless.

$$s_2 = \text{cvtFormatType}(s_1, \text{'unformatted', 'html'})$$

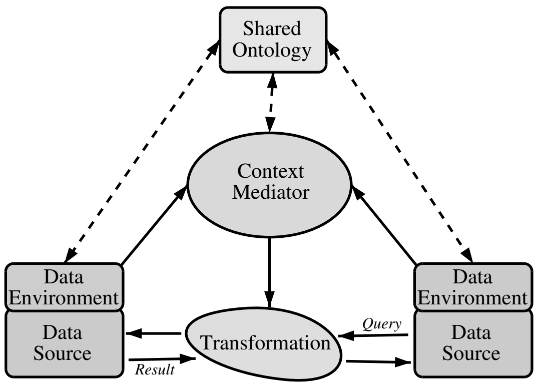


Fig. 3.2: A Semantic Value Architecture

A conversion function is order preserving when two simple values associated with a semantic value do not change their order when they are converted. A conversion function that converts length units, for example, is order preserving. A function converting between code types such as *ASCII* and *EBCDIC* is not order preserving.

Comparisons of semantic values is done with respect to a context, the target context. This requires that properties of context can be resolved. In some instances this is not possible. A resolvable property is one for which the semantic comparison can always be reduced to a comparison of simple values. The *Accuracy* property in the example below is unresolvable and in order for this comparison to be of value a user supplied comparison operator is required. The comparison $v_1 > v_2$ for the two semantic values might intuitively be correct, but some applications may want to treat v_1 and v_2 as equivalent.

$$v_1 = 5.00(\text{Accuracy} = 0.1)$$
$$v_2 = 4.99(\text{Accuracy} = 0.1)$$

[SSR94] propose an architecture that uses semantic values to link heterogeneous information sources. This architecture has five components: information systems, data environments, context mediators, conversion libraries/functions and shared ontologies. Figure 3.2 shows the architecture, its main components and their interaction.

The central component in the architecture is the context mediator which manages all data exchange and is responsible for attribute mapping, property evaluation and conversion. The architecture places no constraints on the data model being used by the component information systems. Each information system component has a data environment which consist of two parts: the semantic-value schema and the semantic-value specification. The first specifies attributes and their properties, while the later specifies values for those properties. The context mediator uses this component to determine whether a data exchange is possible, and if so, what conversions are needed. The conversion functions are maintained by the conversion library. It is used by the context mediator to map information from one system to another.

The shared ontology component specifies *terminology mappings*. The mappings describe naming equivalences between component information systems. The development of an ontology is part of modeling the real world both at a technical and organizational level.

3.4 Summary

In this chapter we have analyzed the concept of federation and looked at different architectures used in federations. The first architecture was the five-level schema architecture. We have highlighted different components used within this architecture, paying particular attention to the schema and processor components. These components are generally the key to resolving semantic heterogeneity. Important in terms of integration was the difference between loose and tight coupling. We have shown that loose coupling is desirable when there are a very large number of generally autonomous systems wanting to communicate. Since this is the case in event federation, this area of research is more relevant to this thesis than solutions that propose tight coupling.

Following the five-level schema architecture we discussed the concept of partial schema unification and mediators. The first architecture discussed gave a detailed description of the different processes involved in mediation. Particularly interesting were the three tasks involved in partial schema unification, namely information discovery, resolution of conflicts and unification. We discussed the tools used for these tasks, the semantic dictionary and the sharing adviser.

We discussed the TSIMMIS project of mediation for the integration of heterogeneous data sources. In this architecture wrappers are used in conjunction with mediator technology. Wrapping services deal with platform-level heterogeneity, while mediator services provide semantic data integration.

Finally, we discussed the different types of heterogeneity that arise between systems. We analyzed why and how they arise. A distinction between fundamental and structural semantic heterogeneity was made. We further analyzed the semantic heterogeneities that arise during system integration and gave a brief overview of how they may be resolved in theory. We described a specific approach used to resolve data ambiguity, using the ideas of *semantic values*. Data that is annotated with semantic values can be compared more easily, making conversions much simpler.

In the next two chapters we use the background information of the previous two chapters to describe the event federation paradigm and an architecture based on that paradigm. The concepts of loose coupling versus tight coupling, wrappers, mediators, semantic dictionary, sharing adviser, data model, conversion functions and export schema are key components in the federated event paradigm and architecture.

4. THE EVENT FEDERATION PARADIGM

Much research has been undertaken trying to standardize database systems, especially their query language and data model. Although recommendations have been made and standards have emerged, it is unlikely that a single, unified database architecture will ever be used. Similarly, it is unrealistic to assume that a common, standardized event notification architecture will evolve. Firstly, different architectures serve different purposes. This has been shown in the area of databases, where *SQL*-based databases are the mainstream architecture used in business applications, while object-oriented databases serve a small niche market. It is the same with event notification architectures. Some event systems aim to be scalable, others more general and flexible. Message-oriented systems focus on security, while newer *publish/subscribe* systems focus on scalability. Secondly, different implementations are currently being used in various environments. The switching cost from one architecture to another is often very high. The need and the desire to share information and resources will nonetheless become greater. Just like documents, images or data records, events are pieces of information that users want to know about and share. General purpose event architectures like the ones discussed in chapter 2 will need to communicate with each other, as well as with other messaging systems. So far, we lack a clear understanding of how this might be done. One possible solution is to get existing event systems wishing to share information to cooperate in the same way as databases or other information sources do in federations.

In this chapter we develop a paradigm for event federations. This paradigm describes the main components of a federation and shows how these interact. We make some minimal assumptions about an event federation and the types of event systems that may participate in a federation. The components we focus on are contracts, domains and gateways. Together, these components form an event federation.

Section 4.1 describes the functionality and aim of an event federation. The section develops an informal specification for the requirements placed on a federation. The specification outlines five characteristics that an event federation must possess. Based on this informal specification, section 4.2 describes the main components of the event federation paradigm. We first discuss the notion of domains and describe their main characteristics. We then explain contracts, which represent an agreement between two or more domains, dictating how they will cooperate. We then look at gateways and their requirements. These components consist of two modules, similar to the wrapper and mediator modules discussed in chapter 3, which together form the translation module of a gateway. Gateways are the connectors of domains in a federation. Section 4.3 shows how the paradigm can be used to build a federation of event federations. This new federation may be built from a mixture of federations and domains. Finally, in section 4.4 we discuss the minimal requirements for an event or messaging system to be able to participate in a federation.

4.1 Requirements

The requirements for an event federation are slightly different than those for a database federation. In the database federation, whether loosely or tightly coupled, the autonomous databases are not linked directly. The federated application is a stand-alone application that breaks-down queries and sends them to the appropriate component databases, before reassembling the data from the individual queries. Posting a query to a mediator possibly returns data from multiple data sources, whereas a query posted to a single component database only returns results from that source. Querying the federation therefore requires an application to be familiar with the interfaces of the mediator. This is an unrealistic assumption for an event system. The clients of an event system tend to be simple applications and it is not realistic to assume that they can deal with multiple interfaces. Therefore, in an event federation it is desirable to allow components to observe non-local event locally. For this, event systems need to be able to establish links with each other that allow them to exchange information. In this way, a subscription to a non-local service from within the local component event system can return notifications from various systems outside the local service. This leads to the first requirement for the event federation model.

1. Heterogeneous event systems must be able to establish links with each other that allow them to observe non-local events locally.

Heterogeneous event systems can be connected in multiple ways. Chapter 3 discussed the difference between loose and tight coupling in a federation. For event systems, loose coupling is a requirement. Rather than having a global authority dictate the integration process, each component event system must be able to determine how it wants to interact with other event systems. The reason for this is two-fold. Firstly, an event federation must accommodate different, heterogeneous event systems with heterogeneous event types. Secondly, different types of integration are possible and often desirable. Event systems must have the flexibility to achieve this. In the five-level schema architecture, multiple ways of loosely integrating component databases was achieved by allowing different users of the federation to create multiple federation schemas. An event federation must support this notion of multiple federation schemas. This is the second requirement for an event federation.

2. The event federation must be able to effect the partial integration of heterogeneous component event systems in multiple ways.

In order for systems to be able to agree on and construct federation schemas, each event system must be able to provide an export schema defining the types of events it is capable of notifying to interested clients. This export schema must be defined using a common data definition language or object model. This is the third requirement placed on the event federation model.

3. Component event architectures must publish an export schema in a common object model.

Unlike a database federation, in an event federation we are not dealing with queries that have to be broken down and sent to the appropriate sources. Instead of queries, subscriptions are sent to a notification service. The notification service replies with the information once an event of interest has occurred. This is simple when an event occurs within the local component event system, but more difficult when an event occurs in a non-local system. It is essential that the event notification, whether it comes from a local source or a non-local one, appears the same to the interested clients. By this we mean, events originating in a foreign domain are translated from the foreign data model of the foreign event system to the data model used by the federation. This leads to the fourth requirement.

4. A component event architecture must be able to translate event instances into an agreed format.

The event federation must support a *register/notify* type of communication model. The publish aspect of the *publish/register/notify* communication protocol has already been mentioned in the previous requirement, which stated that component event systems must publish the event types they can notify in a common data model. The last two aspects of the communication protocol between the components of the federation imply that the communication model must allow clients to subscribe to services and be notified across the boundaries of the component event systems. This is stated by the fifth requirement for an event federation.

5. An event federation must support the *register/notify* style of communication between component event systems.

These are the main requirements we place on an event federation. If multiple event systems are linked and adhere to the above mentioned points, the union of these event systems is called an event federation. We place no other requirements on the system as a whole, nor on the component event systems. Later in this chapter we explain some of the characteristics an event architecture must have in order to be able to participate in an event federation, but for the purpose of describing a federation this is not necessary. In the next section we describe the main components of the paradigm that help fulfill the above mentioned requirements.

4.2 Event Federation Components

We have outlined the main requirements we place on an event federation. The first and second requirements are fulfilled by the first component we describe in this section, the *contract*. Requirements three and four are fulfilled by the *gateway* component. The last requirement is met by a combination of the two components. The third component in the event federation paradigm is the event system itself. We call this a *domain*. Domains establish contracts with each other. A contract indicates an obligation or agreement between two or more entities [HHG90, Hol92]. These agreements are managed by the gateway of a domain.

A federation is formed when two component event systems agree to share information. This agreement comes in the form of a contract. The agreement guarantees each event system that the other participating system is willing and capable of translating its event type into a contract event type as defined by the contract.

A federation is created when at least two domains agree on a contract or when two or more domains join an existing contract. Once a contract has been defined, other domains can join the federation if they fulfill certain requirements. This is determined by a federation administrator, typically an existing domain or set of domains. The administrator is responsible for handing out federation identifiers for each domain wishing to participate in the federation. The identifiers are unique to the federation and are used by the gateways to identify the different domains.

Two possibilities exist for a domain to join a federation. Either it can join an already existing contract, or it can create a new contract with an existing member of the federation. It may also be desirable to allow two new domains, which are not already part of the federation, to join a federation with a new contract. Such a scenario may be desirable when the new contract type is of interest to other members of the federation. This is at the discretion of the administrative domain(s).

Each domain must have at least one gateway to participate in a federation, but may have multiple gateways. A gateway may only participate in one federation, but a domain may participate in many federations via multiple gateways. This also explains how federations can become part of other federations. A gateway can be placed not only at the domain boundary, but also at the federation boundary. A federation with a gateway can take part in another federation.

There may be instances where domains are not capable of providing events to the federation. These domains or applications may only want to receive notifications from other domains. Allowing this is at the discretion of the federation authority. In those instances domains only need to provide *observer functions*, which translate contract types into local types. These functions are necessary when a domain wants to observe contract events locally, whether it has joined a contract or not. Any contract type can have an observer function associated with it and multiple observer functions can exist per contract type. In certain instances an observer type can be identical to a local type, but this is not a requirement.

4.2.1 Domains

Autonomous databases are the building blocks of database federations. *Domains* represent these building blocks in the event federation paradigm. A domain is a logical scope of a collection of event sources where a single administrative power is exercised. The event sources in a domain may be related, but this is not a requirement for membership in a domain. Like the databases in a federation, a domain represents a unit of autonomy. The domain administrator has full control over its event sources and the types defined by each source. Furthermore, the administrator controls who may or may not use the services provided by the architecture. A more formal definition of a domain is given below.

> A domain A consists of a set of event types $T = \{T_{A1}, T_{A2} \ldots T_{An}\}$ defined by sources $S = \{S_{A1}, S_{A2} \ldots S_{An}\}$. E_{A_i} is the set of all event instances associated with the source S_{A1}. A client of domain A is able to receive any event instance of set E_{A_i} by registering interest with a source of set S specifying an event type of set T.

We aim to keep the definition of a domain as minimal as possible. This allows us to incorporate as many different heterogeneous event architectures as possible in an event federation.

4.2.2 Contracts

Central to the notion of event federations is what we call a *contract*. Component event systems wishing to participate in the federation must have a contract with at least one other domain. The notion of a contract is derived from real-life examples where entities interact with each other. Conceptually, a contract is a binding agreement between two or more parties. We use the term contract to mean an agreement between two or more cooperating domains. From a database federation point of view, a contract is similar to the external schema proposed in the five-level schema architecture [SL90]. A more formal definition of a contract is given below.

> Let T_A be an event type defined by a source S_A of domain A, and T_B be an event type defined by a source S_B of domain B. We can define sets E_A and E_B as the sets of all possible instances of T_A and T_B respectively. A contract $\mathcal{C}_{AB}$ between A and B with respect to T_A and T_B is defined as an event type $\mathcal{T}$, the contract type, whose set of all instances is denoted as $\mathcal{E}$, and a set of functions $f : E_A \to \mathcal{E}$ for A and $g : E_B \to \mathcal{E}$ for B.

The function that maps the local event instance to an instance defined by a contract type is called a *schema translation function*. A schema translation function that translates an event instance of type T_A from E_A to an instance $\mathcal{E}$ of type $\mathcal{T}$ is a translation function from E_A to $\mathcal{E}$. Schema translation takes place at the intersection of two domains, the gateway. An event, which is sent across domain boundaries, is translated by the gateway of its domain. The gateway of the foreign domain always receives events conforming to a contract type. The advantage of this model is that applications are easier to build, since they only need to handle the types that they have agreed on with other domains via contracts, making dynamic handling of foreign types unnecessary. Another advantage is that this model prevents the details of an event local to a domain being exposed to a foreign domain.

By allowing domains to establish contracts, we satisfy the first requirement placed on an event federation. Contracts establish a link between component event architectures. Multiple contracts can exist per domain. This fulfills the second requirement, which states that event systems can be integrated in multiple ways. Domains may take part in more than one contract with other domains involving the same local event type. This

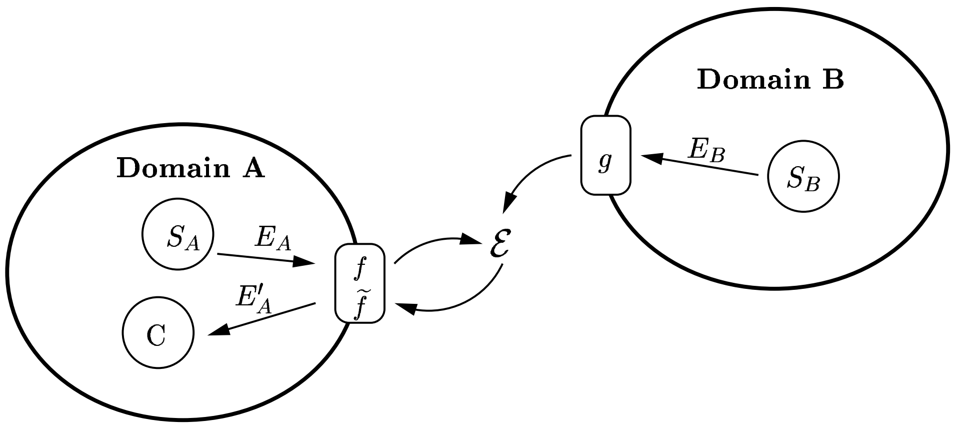


Fig. 4.1: A Contract between two Domains

means that event type T_A of domain A can take part in contract $\mathcal{C}_{AB}$ with domain B and contract $\mathcal{C}_{AC}$ with domain C. The only requirement is that the contract types defined in $\mathcal{C}_{AB}$ and $\mathcal{C}_{AC}$ are not the same. If the two contract types were identical we would have a contract $\mathcal{C}_{ABC}$ rather than two separate contracts.

There are special instances where a schema translation function is *faithful*. By this we mean the function is lossless. In such an instance a non-local event instance can be translated into a local instance via the contract type, using the inverse of the schema translation function provided by the contract. We say that:

> We have a *faithful* function $f : E_A \rightarrow \mathcal{E}$ if there exists an inverse function $\widehat{f} : \mathcal{E} \rightarrow E_A$ such that $f \circ \widehat{f} = I_{E_A}$ and $\widehat{f} \circ f = I_{\mathcal{E}}$ for event instances E_A of type T_A from domain A.

Therefore, if function f is *faithful*, event instances E_B of type T_B from domain B can be translated into instances E_A of type T_A from domain A via $g \circ \widehat{f}$ where $g : E_B \rightarrow \mathcal{E}$. Function $\widehat{f}$ is not part of the contract. We call $\widehat{f}$ an observer function. In the above case, function $\widehat{f}$ is a special observer function because it is the inverse of a *faithful* schema translation function. The purpose of an observer function is to allow local domains to observe contract types locally. Observer functions need not be *faithful*. In general we say:

> An observer function $\widetilde{f} : \mathcal{E} \rightarrow E'_A$ maps instances $\mathcal{E}$ of the contract type $\mathcal{T}$ into instances E'_A of an observer type T'_A. For all *faithful* functions f, if $\widetilde{f} = \widehat{f}$, then $E'_A = E_A$.

In some instances the observer type T'_A may be the super type of T_A for all instances E'_A. This occurs when the observer function $\widetilde{f}$ translates instances $\mathcal{E}$ of the contract type $\mathcal{T}$ into instances E'_A of the observer type T'_A, with the same but fewer attributes than T_A.

Thus far, we have described what we call a *domain-to-domain* contract. This is a contract initiated and created between two domains. In many applications central authority is difficult to establish. As the number of federation members grows, it becomes more difficult to agree on a single set of protocols which satisfy every member. Instead of forcing all participating domains to adhere to a single standard, we take this open approach which allows parties to establish pair-wise agreements with each other. Such contracts are purely localized to the domains in agreement. A domain may have multiple domain-to-domain contracts as we have already stated, and the existence of one contract will not cause conflicts in any way with another contract. This is essential to enable large-scale interoperability. Domain-to-domain contracts do not impose any constraints on the schema translation function, meaning the translation function does not need to be *faithful*.

The second type of contract we allow in a federation is a *global contract*. In certain federated architectures, it is suitable to have an authority which dictates the protocols that cooperative domains should comply with. The protocols the authority defines are called global contracts, and are implemented across the entire federation. A global contract can be used as a super type by participating domains. Domains define their own specialization from the common super type, and all events of a domain-specific type are treated as an instance of its super type by foreign domains. Domains join a global contract by providing a translation function from a local event type to the global contract type.

The differentiation between domain-to-domain and global contracts is only relevant at the time the contract is created. A *hybrid contract* is formed when an existing domain-to-domain contract is joined by another domain. Consider the domain-to-domain contract $\mathcal{C}_{AB}$ between domains A and B. Domain C can join this contract by providing a function $h : E_C \rightarrow \mathcal{E}$ where E_C are all possible instances of the local event type T_C of domain C and $\mathcal{E}$ are all possible instances of the contract type $\mathcal{T}$ defined in the original domain-to-domain contract $\mathcal{C}_{AB}$. Contract $\mathcal{C}_{AB}$ becomes hybrid contract $\mathcal{C}_{ABC}$.

A global contract $\mathcal{C}$ initially has no participating domains. When domain D joins the contract it provides a schema translation function $f : E_D \rightarrow \mathcal{E}$ for all instances E_D of event type T_D to instances of the contract type $\mathcal{T}$ defined in $\mathcal{C}$. Contract $\mathcal{C}$ becomes contract $\mathcal{C}_D$. If domain F now joins this contract as well, again providing the appropriate translation function $g : E_F \rightarrow \mathcal{E}$ for all instances E_D of event type T_D to the contract type $\mathcal{T}$ defined in $\mathcal{C}_D$, this contract becomes a domain-to-domain contract $\mathcal{C}_{DF}$.

The distinction between global and domain-to-domain contracts allows for a more tightly or more loosely coupled system, depending on the need of the application. Both contract types can coexist within a federation, giving the system the flexibility to integrate different domains at different granularity. Unlike in tightly coupled database federations where the administrative authority usually imposes and global schema which participating databases must comply with, this paradigm allows for less restrictive cooperation of domains.

4.2.3 Gateways

A *gateway* is the physical component that *connects* domains in an event federation. It performs two basic operations. It translates objects from a local representation to a global representation and vice versa. It does this on two types of data, notifications and subscriptions. The two operations can be further broken down into two sub-operations. For notifications, the first sub-operation performs a translation on an instance of a local type represented in the local data model to an instance of a local type represented in the global data model. The second sub-operation translates an instance of a local type represented in the global data model to an instance of a global type (the contract type) represented in the global data model. These sub-operations are also performed in the other direction. The concept is the same for subscriptions. These are translated from subscriptions given in the local representation for a local type to subscriptions for a global type represented in the global data model. This is again performed by two sub-operations.

In the previous section we talked about schema translation functions. By breaking down the translation function into two sub-functions, we identify the schema translation function provided by the contract as the function which translates event instances from the contract type into instances of the local type (the observer type) represented in the global data model. This function therefore resolves structural semantic heterogeneity, whereas the second translation function resolves the conceptual data definition language mismatch [HM93].

It is convenient to view a gateway as consisting of two separate modules. The first module performs the translation from the local to the global data model and vice versa, while the second performs the translation from the local type to the global contract type and vice versa. In chapter 3 we introduced the mediator/wrapper architecture.

The functionality of a gateway can be separated along those lines. The wrapper module translates the information into a common representation understood by the federation, while the mediator module is responsible for resolving semantic heterogeneity by providing schema translation functions.

Gateway Event Translation

Within a domain, a gateway appears to other entities as an event consumer when it is handling local events that are part of a contract with a foreign domain. Local entities (event producers or mediators) send local event instances to the gateway for translation and then transmission to other domains. A local event is translated into the global data model by the wrapper and into the contract type by the mediator, using the schema translation function of the contract, before being sent to other domains taking part in the contract. Assuming domain A is part of a contract we say:

> Gateway G_A of domain A with a contract $\mathcal{C}_{AB}$ performs the functions $f_1 \circ f_2 : E_A \rightarrow \mathcal{E}$ for all instances E_A of type T_A, where $f_1 : E_A \rightarrow E_{A_g}$ and $f_2 : E_{A_g} \rightarrow \mathcal{E}$, with E_{A_g} representing all instances of the local type T_A in the global data model and $\mathcal{E}$ representing all instance of the contract type $\mathcal{T}$ of contract $\mathcal{C}_{AB}$.

Assuming the local domain taking part in the contract has provided an observer function for the contract type, events coming from a foreign domain arrive at the local gateway as an instance of the contract type. This contract type is translated into instances of the observer type represented in the global data model via the mediator using the observer function and into the local data model via the wrapper, before it is passed to the event consumers within the local domain. In such instances the gateway appears to entities of the local domain as an event producer. Assuming domain A and B are part of a contract and A has provided an observer function, we say:

> Gateway G_A of domain A with a contract $\mathcal{C}_{AB}$ for the local event instances E_A of type T_A and an observer function g_1 for the contract type $\mathcal{T}$ performs translations $g_1 : \mathcal{E} \rightarrow E'_{A_g}$ and $g_2 : E'_{A_g} \rightarrow E'_A$ on all instances $\mathcal{E}$ of type $\mathcal{T}$ defined in $\mathcal{C}_{AB}$ where E'_A are all instances of the observer type represented in the local data model and E'_{A_g} are all instances of the observer type represented in the global data model.

We note that for a *faithful* function f_2 of contract $\mathcal{C}_{AB}$ we can choose $g_1 = \widehat{f_2}$ so that $E'_A = E_A$.

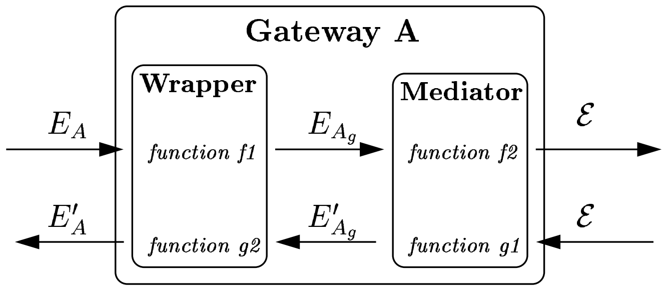


Fig. 4.2: Event Translation at the Gateway

This fulfills the fourth requirement for a federation. By providing an observer function g_1 a domain is able to translate and represent non-local event instances within the local domain using the local data model.

Once translated, a gateway either sends events to components within the local domain or to other gateways within the federation. Events coming from foreign domains as instances of the contract type are translated and then forwarded to event sinks (clients) within the local domain. How events are forwarded depends on the local event system. Local event instances are translated into the contract type and are then forwarded by the gateway to all domains having registered interest in that event.

Figure 4.2 illustrates the event translations a gateway performs. Besides translations, the gateway also manages all contracts that exist between its domain and foreign domains. This means a gateway is responsible for determining which function to apply to which event instance. This applies to incoming as well as outgoing event instances.

Gateway Event Publication

A gateway must publish the event types it is willing to share with other domains using the data definition language of the federation. Each domain can choose which local event types to publish and thus make available to members of the federation. The local event definitions must be translated into definitions using the global definition language of the federation. These translation functions are maps between data definition languages. More formally, we say:

> For the set of local event types $T_{loc} = \{T_{A1}, T_{A2}...T_{An}\}$ of domain A, there exists a function f_{schema} which translates each element of T_{loc} to an element of T_{glob}, where T_{glob} is the set of all local event types represented in the global data definition language.

Besides local event types, a gateway also publishes the contract types of the contracts it has taken part in. These do not need to be translated because the contract types are represented using the global data definition language. Also publishing the contract types allows other domains to determine whether they want to join an existing contract with a domain or establish a new contract. This differentiation is important for two reasons. Firstly, it gives domains the flexibility to establish contracts that are best suited to their requirements. A contract type suitable to one domain may not be suitable to another. Secondly, it provides a means of managing translation functions more efficiently. In case a contract is agreed upon for an existing contract type with a foreign domain, the translation function for translating from the foreign type to the local type is not required. In case a contract is established with a non-contract type, the translation functions must be provided as part of the new contract.

This fulfills the third requirement placed on a federation. The translation function f_{schema} enables a domain to publish the event types it is willing to share with the federation using the data definition language of the federation.

When a domain provides an observer function for a contract type, it may be desirable for the gateway to translate the observer type into an observer type represented using the local definition language of that domain. In this way the observer type can be published within the domain. This is not a requirement because in some instances domains do not want to allow local clients to receive foreign events.

Gateway Subscription Translation

Besides translating event instances and event type definitions, a gateway must handle subscription translations. A subscription from the local domain arrives at the local gateway in the subscription language used by that domain. The gateway appears to components within the domain as an event producer, with which components register. The subscription needs to be translated into a subscription for an instance of an observer event type represented in the global data model. The observer event type in the local event system is E'_A and the same event instance represented in the global data model is E'_{A_g}. Assuming we have the observer functions $g_1 : \mathcal{E} \rightarrow E'_{A_g}$ and function $g_2 : E'_{A_g} \rightarrow E'_A$ for translating a contract type into an observer type in the local data model, we require a set of translation functions for subscriptions for event instances of the observer type to subscriptions for event instances of the contract type $\mathcal{T}$. More formally, we say:

Gateway G_A of domain A with a contract $\mathcal{C}_{AB}$ and an observer function

> for the contract type, must perform a set of translation functions $f_{sub1} : S'_A \rightarrow S'_{A_g}$ and $f_{sub2} : S'_{A_g} \rightarrow \mathcal{S}$, where subscription $\mathcal{S}$ is a subscription to instances $\mathcal{E}$ of contract type $\mathcal{T}$ defined in contract $\mathcal{C}_{AB}$, S'_{A_g} a subscription for event instances E'_{A_g} of the observer type T'_{A_g} represented in the global data model and S'_A a subscription to event instances E'_A of the observer type T'_A represented in the local data model.

Once translated, the subscription is passed to the gateways of the domains with which the local domain has a contract for the event type of the subscription.

The above definition satisfies outgoing subscriptions. A gateway must also translate incoming subscriptions. An incoming subscription arrives at the local gateway as a subscription for a contract type. A subscription for a contract type must be translated into a subscription for a local type. More formally, this means:

> Gateway G_B of domain B with a contract $\mathcal{C}_{AB}$, must perform a set of translation functions $g_{sub1} : \mathcal{S} \rightarrow S_{A_g}$ and $f_{sub2} : S_{A_g} \rightarrow S_A$, where subscription $\mathcal{S}$ is a subscription to instances $\mathcal{E}$ of contract type $\mathcal{T}$ defined in contract $\mathcal{C}_{AB}$, S_{A_g} a subscription to event instances E_{A_g} of the local type T_{A_g} represented in the global data model and S_A a subscription to event instances E_A of the local type T_A represented in the local data model.

Once translated, the subscription is passed from the gateway to a local component handling subscriptions within the domain. Depending on the component event system, this can be a broker or an event source. The gateway component acts as an event client within the local domain by subscribing to an event service.

This fulfills the fifth requirement placed on an event federation. Through the use of subscription translations via the contract type, a gateway is able to support the *publish/register/notify* style of communication between domains.

Security

Although it is not a strict requirement, a gateway should provide security for the local domain. It should only deal with domains that are approved by a federation authority. Further security may come in the form of a role access model such as OASIS [Hay96] (see chapter 7). The degree of security provided by the federation differs depending on the type of federation.

A very simple form of security provided by this model is the ability for a domain to

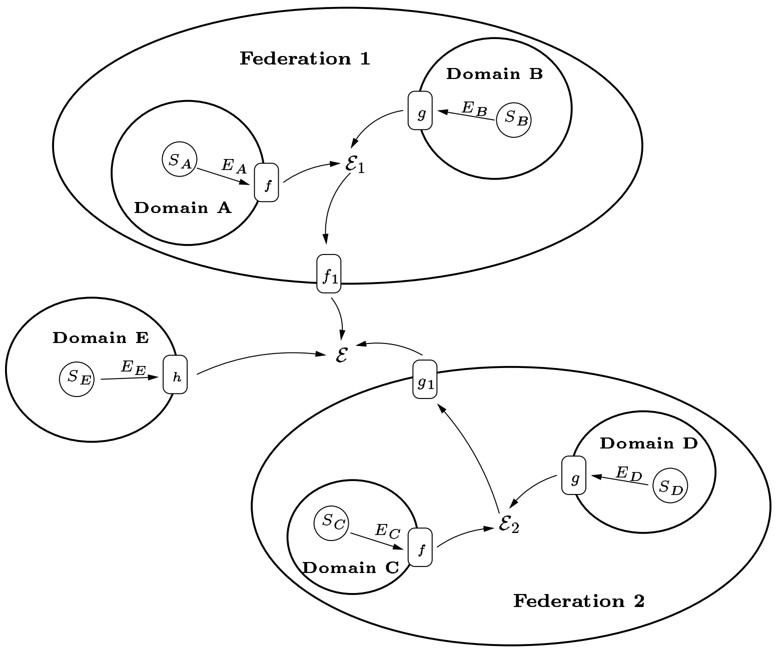


Fig. 4.3: A Federation of Federations

choose which event types it is willing to share with the federation. Further possibilities might include limiting the number of contracts for a domain (or event type) or limiting the number of subscriptions. Even though there are no strict rules or requirements, the event federation paradigm provides the application designer with the ability to implement security features at the gateway.

4.3 A Federation of Federations

Using the event federation paradigm developed in this chapter, one can build new federations from existing federations. The new federation can contain federations as well as individual domains. Figure 4.3 illustrates the notion of contracts between federations and domains.

The paradigm does not need to be extended to handle these types of federations. A gateway module must be placed at the boundary of a federation. Federations can then create contracts with each other or join existing contracts. The gateways at the boundary of a component federation provide the services that were described in the previous section. A gateway manages contracts between federations, handles event translations for instances of contract types and observer types, translates subscriptions and publishes the event types the component event federation is willing to share with other members of the newly formed federation.

A client in domain A in figure 4.3 can receive event instances generated in domain C if domain A provides an observer function for instances $\mathcal{E}_1$ and *Federation 1* provides an observer function for instances $\mathcal{E}$. The client subscribes to the observer type. The subscription is translated at the gateway of domain A and at the gateway of *Federation 1*, before it is passed to the gateway of *Federation 2* where it is again translated by the two gateways. When an event, which matches the subscription, occurs in domain C it is translated by the gateway of domain C into an instance $\mathcal{E}_2$ and then by the federation gateway into an instance $\mathcal{E}$. From there it is passed to the gateway of *Federation 1*, where it is translated into an instance $\mathcal{E}_1$. Finally, the gateway of domain A translates it into an instance E'_A of the observer type.

By placing a gateway at the boundary of a domain or federation, we are effectively able to hide the details of the domain. In that sense, a federation with a gateway cannot be distinguished from a domain with a gateway. Our previous definition of a domain consisting of a set of event types T is therefore valid for a federation. If the component federations making up the new federation use the same global data model, the gateways of the component event federations do not require the functionality to translate between data models.

4.4 Assumptions

No assumptions have been made about the event systems that make up a domain. The aim is to keep the paradigm general, to allow it to accommodate as many different types of event systems as possible. Nonetheless, we do need to make some assumptions about the event systems making up a domain.

Firstly, an event system wishing to be part of an event federation must have a publishing service and the ability to describe its events in a schematic manner. Here we do not make any assumptions about how the published information is presented, or the event model in general. The autonomous databases of a federation tend to have very different data models, but they all have a mechanism whereby they describe the information that the database holds. For databases this is the conceptual schema. We term this first requirement placed on an event system to allow it to participate in a federation *conceptual event representation.*

The second requirement is that an event system provides functionality that enables participants to register or subscribe to a service. This means it must provide a registration service whereby entities interested in event notifications can register for a particular event. We call this requirement *register/notify support* for event systems. How this registration happens in detail is not relevant. More sophisticated event services have allowed registrations to include wildcard values [Spi00, MB98, BBMS98, BBHM96, RW97, CRW00] for parameterized filtering on event occurrences. This is not necessarily a requirement. We feel that filters can be implemented at the domain boundary. This is important because otherwise we would exclude a number of event architectures from participating in the federation.

4.5 Summary

This chapter described the event federation paradigm. We outlined the requirements for such a federation, specifying five characteristics which must be fulfilled by a group of cooperating event systems before they can call themselves an event federation as we define it. These requirements are:

1. Heterogeneous event systems must be able to establish links with each other that allow them to observe non-local events locally.
2. The event federation must be able to integrate heterogeneous component event systems partially in multiple ways.
3. Component event architectures must publish an export schema in a global object model.
4. A component event architecture must be able to translate event instances into an agreed format.
5. An event federation must support the *register/notify* style of communication between component event systems.

We then described the major components of the paradigm that help satisfy the five requirements. We partitioned component event systems into domains, where each domain consisted of a set of event sources. The contract component was central to linking different domains in a loosely coupled system. The gateway component, consisting of a wrapper and mediator module provides the glue between domains in the paradigm. It manages contracts and handles event and subscription translation. Furthermore, it publishes the event types available within the domain. These modules are the building blocks of an event federation.

Using the event paradigm, we showed how new event federations can be formed from existing federations by providing gateways at the federation boundary. A federation with a gateway appears to other entities in a federation as a domain consisting of a single component event system.

We discussed two main requirements for an event notification architecture to be able to participate in a federation. Firstly, it has to have a type system for describing its events. Secondly, it has to have subscription and notification facilities. We called these requirements *conceptual event representation* and *register/notify support*.

5. AN EVENT FEDERATION ARCHITECTURE

This chapter describes the architecture of a gateway component, which links heterogeneous event systems based on the paradigm developed in the previous chapter. The architecture provides an open, event-based platform for building large distributed applications by linking existing event architectures in a loosely coupled, federated manner. The emphasis is on *interoperability* among component event systems. The design of the gateway component is based upon the following principles:

Openness One characteristic of distributed systems is that there is usually a diversity of machines, operating systems, communication protocols and middleware platforms. The architecture must therefore be open and have clear interfaces for it to easily integrate heterogeneous environments.

Language-independence Since it is impossible to have a single, unified platform to satisfy everyone and equally unrealistic to persuade people to use a single programming language, the design aims to achieve language-independence.

Extensibility The architecture must allow for customization and extensibility, in order to suit the different operating environments. Local extensions should be domain-specific and should not affect the system's interoperability with other domains.

The architecture makes extensive use of XML technology. This is a major step toward achieving the design goals and interoperability. It guarantees openness, language-independence and extensibility. The extra overhead of having to parse XML is insignificant in this context. The system interfaces are defined in XML, which can be delivered across any transport protocol capable of sending text, imposing a minimum restriction on the requirement of the communication protocol. Moreover, XML is programming language independent. It does not rely on language mappings to achieve language independence. Potentially, any programming language that can send a text string across the network can fit into the architecture.

This chapter begins by describing the use of XML technology in the federated architecture for event publication, notification, subscription and translation. It explains the use of XML Schema [Con01a, Con01b, Con01c] for event publication, XML [Con00] for event notification, *FedReg* for event subscription and XSL Transformations (XSLT) [Con02c] for event notification and subscription translation. Section 5.2 describes *FedReg*, a federation-level subscription language based on XML, used by the federation. It is a versatile language that aims to express as many subscription language styles as possible. In section 5.3 we describe the components of a gateway and describe their interaction. Besides the wrapper and mediator, we introduce a storage layer, two communication components (one for domain-level communication and one for federation-level communication) and a subscription filter. We then begin a more detailed discussion of each of these components. In section 5.4 we describe HERMES, an event-based messaging architecture. It is an architecture well suited for large-scale distributed applications. HERMES is used for connecting gateways in the federated event architecture. Section 5.5 discusses the implementation of the mediator module of the gateway component. This module is shared by all gateways. It is responsible for resolving structural semantic heterogeneity between domains. It uses the translation functions of the contract to translate event instances between the contract type and the local event type using the global data model. Section 5.6 describes the wrapper module of a gateway. This module is unique for each gateway and its implementation depends mainly on the event architecture of the domain. We look at a specific implementation of a wrapper for the HERALD event architecture. In section 5.7 we discuss the contract creation process and a tool developed for this purpose. Using the export schemas of a domain, the tool creates a contract consisting of a contract event type and the corresponding translation functions. We also discuss cross-domain event publication and show how domains can query other domains for event services through gateway components. Section 5.8 describes how a domain can integrate contract types into the local domain by providing observer functions.

5.1 Overview

An event federation is formed when two domains agree to share information. This agreement exists in the form of a contract. A contract between two or more domains contains a contract type and translation functions from local types to the contract type, for each domain taking part in the contract. Contracts are managed by gateways at the domain boundary. Besides managing contracts, a gateway also publishes domain specific event types using the global data definition language and translates subscriptions and notifications. The architecture also provides a mechanism for creating contracts. Along with a contract a domain can provide an observer function for a contract type.

This allows the domain to observe a contract type within the local domain. These observer functions are managed by gateways. In this section we describe how this is implemented using XML Schema, XML, and XSLT.

XML serves as the foundation for inter-operation among the heterogeneous domains in the event federation architecture. The rationale behind the choice of XML is as follows:

- It is open and extensible, which enables features to be added to our architecture without breaking the interoperability goal. Local customizations can be made by applications without affecting the federation.
- It uses a text-based representation which imposes few restrictions on the application implementation. Applications can be programmed in different programming languages and any transportation protocol capable of sending text can be used.

We use XML in the architecture to represent event instances, and XML Schema to define the event types that describe those instances. The *contract type* defined in a contract between domains is also defined using XML Schema. The use of XML to represent events naturally leads to the choice of XSL Transformation (XSLT) for the implementation of schema translation functions. XSLT is designed for the generic transformation of XML documents. The second part of a contract is thus a set of XSLT documents which contain the translation functions that map local event types to contract types. The observer function is also an XSLT document. It defines a mapping for translating a contract type into a local observer type.

The architecture defines a minimal schema for representing event instances. It provides a definition for a base event type, designed to serve as the super-type of all events. The definition for a *BaseEventType* in XML Schema is shown in figure 5.1(a). The base event has parameters for holding essential information about an event, including its time of creation, an identifier and its source of origin. The source of origin is used by a gateway to match-up contracts with event instances. The origin of an event instance, together with the event type information, determines which translation function the gateway uses to map an event instance. The federation extends event types from the base type using the standard XML Schema *derivedBy* attribute. Only derivation by extension is currently supported.

Conflicts in parameter names are not permitted. This becomes difficult to enforce once a federation, and hence the event hierarchy, grows large. We adopt XML Namespace [Con99] to solve this problem. Essentially, a federation defines a namespace of its own and uses it as a prefix for all contract types it defines. Conflicts are also not permitted

```
<complexType name="BaseEventType">
 <element name="id" type="xsd:long"/>
 <element name="src" type="xsd:string"/>
 <element name="ts" type="xsd:timeInstant"/>
</complexType>
```

(a) The Base Event Type

```
<complexType name="GPSEventType">
 <complexContent>
  <extension base="BaseEventType">
   <element name="longitude" type="xsd:long"/>
   <element name="latitude" type="xsd:long"/>
   <element name="altitude" type="xsd:long"/>
  </extension>
 </complexContent>
</complexType>
```

(b) A Contract Type

Fig. 5.1: Event Type Definitions

for event publication. Each domain, when publishing the event types it is willing to share with the federation, has its own namespace. The namespace for domains is managed by a federation authority.

The first part of a contract is a contract type defined in XML Schema. The contract type is either directly or indirectly derived from the *BaseEventType*. Figure 5.1(b) shows an example contract type named *GPSEventType* derived from *BaseEventType*. The base event type is shown in figure 5.1(a).

A gateway performs event translations, mapping local types to contract types and possibly contract types to observer types. An event source signals event occurrences to interested clients. A gateway acts as a client within the local domain in instances where a cross-domain subscription for the local event type exists. The local source passes a local event instance to the gateway of the domain. Upon receiving the event from the source, the gateway maps it to the contract type. It uses the XSLT mapping provided by the contract and performs filtering on the translated event (this is discussed below). A gateway acts as an event source for the contract type from the perspective of other domains. The contract type is passed to domains for which the filtering expression is matched. The gateways in other domains, acting as clients for contract events, perform a translation from the contract type to the observer type. Once translated, the observer type is passed to the local client from which the federation-level subscription originated.

A federation-level subscription is a subscription for an observer event type. It originates in a domain which has provided an observer function for a contract type. The subscription is submitted to multiple cooperative domains which are joined by a contract. Federation-level subscription is also known as a cross-domain subscription. A client expresses interest in terms of an observer event type defined by the observer function. The subscription is submitted to the gateway of the local domain where it is translated and then propagated to the correct foreign domains. The mapping function for subscription translation must be provided by the individual domains. There are two mappings. The first maps a local observer type subscription to a contract type subscription. The second maps a contract subscription into a subscription for the local type that is part of the contract in the foreign domain. From a client's point of view, the subscription for an observer type is the same as a subscription for a local event type. This means that the gateway appears as an event source or broker (depending on the component event architecture) to clients registering interest with an observer type. Clients cannot necessarily distinguish between a local type and an observer type.

There are instances where a component event architecture does not support source-side filtering or fine-grained subscriptions. Since we want to filter subscriptions as close to the source as possible, a gateway must provide a module for this. This module supports fine-grained filtering and is provided by the gateway of the domain for which the subscription is intended. Subscriptions are translated into type subscriptions before being passed to the component event system. This makes cross-domain subscriptions more complex because part of the filtering is managed by the component event system and the other part by the filter module of a gateway itself. This is caused by the difference in subscription styles used by event notification architectures. We have developed an XML-based subscription language, *FedReg*, which can express a wide variety of subscription styles. This has complicated the task of subscription translation and for this reason the gateway must perform filtering. Translating from a less expressive to a more expressive subscription language is possible, but the opposite is not without providing filtering.

Mapping from a more complex to a simpler subscription language means information loss occurs if no measures are taken at the point of translation. In our architecture, this is managed by the subscription filter of the gateway. Since the aim is to filter as close to the source as possible, the filtering is performed by the gateway closest to the source. The architecture translates subscriptions at the wrapper and mediator level in the home domain, while the foreign domain provides filtering. The foreign domain only performs a simple translation, mapping an attribute-based subscription to a type subscription of the local event system. The original contract subscription is stored. After the translation, the subscription is in a format that can be accepted by any domain. The gateway uses this to register interest with a type, using the type subscription. When an event of interest occurs it is passed to the gateway, where the subscription is retrieved and the event instance filtered by the subscription filter. A

```
<xce:Subscription>
  <xce:Template type="GPSEvent">
    <longitude>0:07:00E</longitude>
    <latitude>52:12:00N</latitude>
    <altitude>*</altitude>
  </xce:Template>
</xce:Subscription>
```

(a) Registration Example I

```
<xce:Subscription>
  <xce:Template type="ActiveBadgeSighting">
    <room xce:filter="='111' || ='222'"/>
    <time xce:filter="< 18:00 && > 09:00"/>
  </xce:Template>
</xce:Subscription>
```

(b) Registration Example II

Fig. 5.2: FedReg Examples

more thorough treatment of these and other issues is presented in sections 5.2 and 5.5.

5.2 The Federation-level Registration Language

This section describes *FedReg*[1], the federation-level subscription language. We describe *FedReg* by looking at some example subscriptions. The examples provided in figure 5.2 are based on a location sensor application. The first example is a subscription for a GPS event, the second a subscription for an *Active Badge* sighting in a room.

FedReg has a namespace URI `http://www.cl.cam.ac.uk/2002/FedReg`, and all entities it defines exist in this namespace. In the example shown in figure 5.2, we give the namespace an alias `xce` for brevity.

The document element of every *FedReg* document is named `Subscription` and it contains one or more elements named `Template`. A `Template` element has an attribute `type` whose value specifies the event type that this template is based on. The type name of an event template is the name of its corresponding event type. An instance of an event template is similar to an instance of an event, but it may contain additional attributes and elements to indicate filtering constraints and meta information.

5.2.1 Pattern Matching

Specification of the conventional pattern-matching style of registration is expressed by filling parameters of event templates with concrete values or a wildcard character (*). The registration for `GPSEvent` in figure 5.2(a) illustrates this type of registration. The

[1] *FedReg* has been developed together with Walt Yao.

registration specifies an interest for any GPS sighting at longitude $0^o7'0''$ East and latitude $52^o12'0''$ North, at any altitude.

5.2.2 Filtering Expressions

We have designed a simple template filtering language, which allows specification of complex filters involving equality and inequality, logical operators, and regular expression matching on strings. Specification of filtering expressions is through a special attribute defined by the event template, named `filter`, which exists in every parameter of an event type.

The filtering language employs a notation where the left-hand-side operand of some binary predicate is left out. This is because we expect that the left-hand-side operand is always the name of the parameter itself for these binary predicates. This does not apply to unary predicates. For example, a side effect expression evaluates a user-defined function and returns a boolean result. Figure 5.4 shows binary operators in the template filtering language and their precedence. The higher the precedence, the earlier the operator will be evaluated.

The interest for `ActiveBadgeSighting` events in figure 5.2(b) demonstrates the use of filtering expressions. Filtering expressions are given to the parameter `room` and `time`, which collectively indicate the interest in the presence of any person in room `111` or `222` between `0900` and `1800` hours.

5.2.3 Meta Information

On a conceptual level, a registration includes specification of interest in terms of event templates and filters. However, from an engineering perspective, we also wanted to include meta information relating to a registration so that it can be conveniently applied if desired. The meta information may include disposal rules, re-registration instructions, quality of service requirements and scope specification.

Disposal rules allow a registration to be discarded automatically by an event service according to specified conditions. Conditions may include time of day, change of security attributes, and change of execution environment. A client can express arbitrary conditions by providing a script, while we have also provided a number of built-in functions for common conditions.

Re-registration instructions allow a discarded registration to be re-registered. This

```
<xce:Subscription>
  <xce:Template type="LoggedOn">
    <xce:meta type="disposal">18:00</xce:meta>
    <xce:meta type="register">09:00</xce:meta>
    <user>*</user>
    <host>*</host>
  </xce:Template>
</xce:Subscription>
```

Fig. 5.3: A Subscription with Meta Information

complements the disposal rules so that registration of interest can be made automatic. A client needs to submit recurring interests only once. For example, for an audit logger which only needs to keep logs for logged-on users during office hours, a single registration, shown in figure 5.3, can be used.

Precedence	Name	Operators
1	Relational Operator	<, >, <=, >=
2	Equality Operator	=, !=
3	Logical AND Operator	&&
4	Logical OR Operator	||

Fig. 5.4: Operators for Template Filtering

A registration can also be attached with a specification for quality of service requirements. This allows event services that have the capability to prioritize events for dispatching, based on hints provided by clients. For critical events, such as network failure notification, a client may specify real-time QoS requirements, effectively requesting the event service to dispatch such an event as soon as it has occurred. Less critical events, such as those sent to a logger for auditing purposes, can be dispatched at a time more convenient to the event service. Even though it is argued that events are primarily used for real-time notification, the specification of quality of service is nonetheless useful, especially when an event service has to compromise its real-time responsiveness due to a heavy load.

The extra functionality provided by *FedReg* will most likely not be supported by the component event systems. The gateway component must be able to manage this extra functionality. For example, the re-registration example above is unlikely to be supported by component event systems. Therefore, the gateway must interpret meta information and apply de-registration and re-registration, if this feature of *FedReg* is to be supported.

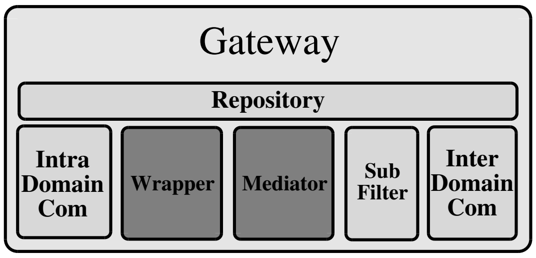


Fig. 5.5: A Gateway Component

5.3 The Gateway Architecture

We have previously seen that the main components of a gateway are the wrapper and mediator. The wrapper component resolves data model mismatches, while the mediator component resolves structural semantic heterogeneity by mapping local types to contract types via the schema translation function and contract types to observer types via the observer function. This architecture now needs to be extended to include the communication between the gateway and the local domain on the one hand and the local gateway and other gateways on the other hand. Furthermore, the gateway needs to handle subscription language heterogeneity.

Therefore, besides the translation modules, gateways consist of two communication modules, one for communication with gateways of other domains and one for communication with components within the local domain. A gateway also has a storage module, which stores contract types, translation functions and observer functions. Finally, a gateway also has a filter component which resolves the heterogeneity of subscription styles between domains. Figure 5.5 shows the basic architecture of a gateway. The intra-domain communication depends on the architecture of the component event system and is handled by giving the gateway the functionality of an event client (source and sink) of the local domain.

The communication between gateways is managed by the inter-domain communication layer. This module is based on HERMES, a content-based notification architecture, which is discussed in more detail in section 5.4. Gateways communicate via XML-defined messages. The HERMES architecture uses four message types for communication. The extra requirements of a federation has meant that four extra message types had to be added to the system. XML Schema was used to define the different message types (see appendix A). The messages are XML objects (strings) with a structure defined by the schema. The eight messages used for inter-domain communication are:

Type Messages These messages add new event types to the system. The message contains the type information (schema) for an event type. This information is stored at special brokers. Type checking occurs against this schema.

Advertisement Messages These messages indicate an event producer's ability to publish a specific event type. These messages are used to set up dissemination paths in the network of brokers (gateways).

Subscription Messages These messages are used by subscribers to indicate interest in a certain event type. They are used to set up paths for the events to follow through the network for event notification.

Publication Messages These messages carry the event instances that have occurred at the publication node. They follow the paths created by the advertisement and subscription messages and determined by the dissemination trees.

Contract Messages These messages are used to inform a gateway of the appropriate translation functions for a specific contract. As mentioned, the gateway manages all translation functions of the contracts that its domain is involved in. The contract message contains a schema translation function and a subscription translation function.

Observer Messages These messages are used to inform the gateway of the observer type and observer function for a contract type. The message contains both a translation function from the contract to the observer type and a translation function for the subscription.

Event Discovery Messages These messages enable a domain to query other domains about their local event types. A *discover* message prompts the gateway to produce a *discover reply* message containing the requested information.

Discovery Reply Messages These messages contain schema information for all the event types the domain is willing to share with other domains. This includes contract type definitions.

Before discussing the components of a gateway individually, we describe the gateway architecture by showing how the different components interact. Components within the federation communicate using the messages described above. Here we look at some of the possible states of a gateway component when receiving different types of messages from other domains and from the local domain. We consider the following scenarios:

- A domain joins a federation.

- A discover message is received.
- A contract message is received.
- A subscription message is received from a foreign gateway.
- A subscription message is received from the local domain.
- An event notification message is received from a foreign gateway.
- An event notification message is received from the local domain.

5.3.1 Joining a Federation

A domain must be able to uniquely identify itself within a federation. It does this using an identifier it receives from the federation authority prior to joining the federation. This is used by the gateway to tag event notifications and subscriptions to indicate their domain of origin. Furthermore, the domain receives its own unique namespace for event publication. Once these two requirements are fulfilled, a domain can join a federation. It has not formally joined a federation until a contract has been created involving one of its event types. Before this can happen, a gateway must publish the local event types it is willing to share with the federation. This process is initiated by the intra-domain communication component of the gateway. It acts as a client application within the local domain by discovering the availability of event services. The exact detail of this depends on the component event architecture. One possible scenario is for the gateway to query an event broker or advertising service within the component event system, which contains event service information for that domain.

Once this information is available at the gateway, the wrapper component performs metadata translation. Since the data definition language (DDL) of the federation is XML Schema, all local event definitions are translated into XML Schema definitions. These definitions are stored in the repository of the gateway for subsequent use.

Once this information has been published, other domains can query the gateway for available event services. This is done in the form of a *discover* message. The gateway responds with a *discover reply* message which contains all the event definitions. This information is then used by other domains (or contract creation tools) to create contracts. Alternatively, the local domain can query other domains for available event types to create a contract with.

Once a contract has been created, the local domain receives a type message and a contract message with the appropriate contract information. The first message informs the domain of the new contract type by specifying the event definition. The second message contains the necessary translation function to map between the local and the contract type. The gateway stores this information and is now fully integrated into the federation.

At this point an observer function can be provided to enable the local domain to observe the contract type locally. Once this has been done for some or all contract types, the domain can subscribe to observer types and become a fully participating member in the federation.

5.3.2 A Discover Message

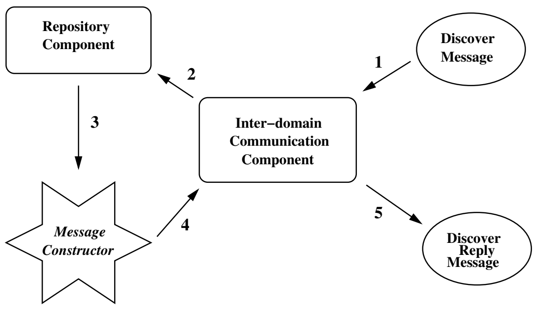


Fig. 5.6: Handling a Discover Message

A *discover* message is a query, asking the local domain to identify all event types it is willing to share with the federation. The reply to this message is always a *discover reply* message. Upon receiving a discover message, the gateway repository is queried. The repository holds information about local and contract types. With this information, the gateway constructs a *publication* message containing all the event types and contract types. The domain, from which the discover message originated, receives the results from all domains contacted in the form of discover reply messages. Some of the contract types will be received in duplicate by the domain. The gateway from which the query originated handles this be merging the messages and indicating that the same contract types exist in multiple domains.

Each gateway maintains a list of domain members and their gateways. This allows a gateway to send discover messages to all members of the federation. This list is used both when sending out discover messages and when receiving publication messages. In the latter case it is used to ensure that all domains have replied.

Generally, discover messages are used within the federated architecture for contract creation. This means that not only other domains send discover messages, but also contract creation tools (see section 5.7).

5.3.3 A Contract Message

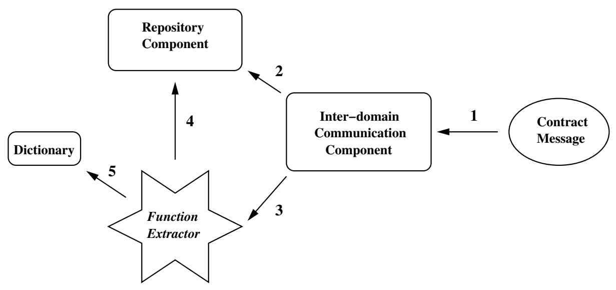


Fig. 5.7: Handling a Contract Message

A contract message provides the gateway with the translation function for a contract type. When receiving a contract message the gateway first checks whether the contract type referred to in the message exists in the repository. Prior to a contract message, a gateway will have received a type message informing it of the newly created contract type. This is necessary so that the domain can set up the dissemination tree required for routing contract types within the federation. The gateway will have extracted the type information from the type message and stored it in the repository. If the contract type is found in the repository, the translation function is extracted from the contract message and stored in the repository.

The gateway maintains a dictionary matching event types to translation and observer functions. This dictionary also needs to be updated when a contract message is received to enable the gateway to translate event instances correctly. When a contract message is received, a dictionary entry is added that links the event type to the translation function.

The dictionary is used for local to contract type and contract to observer type translation. A publication message from a foreign domain contains a contract identifier. This identifies the contract type and enables the gateway to retrieve the correct observer function. When a local event instance is received by the gateway it must also be mapped using the correct function. The correct function is found by searching in the dictionary for all available translation functions for that type and by determining for which contract types a subscription exists. This information is also maintained by the gateway dictionary.

5.3.4 A Foreign Subscription

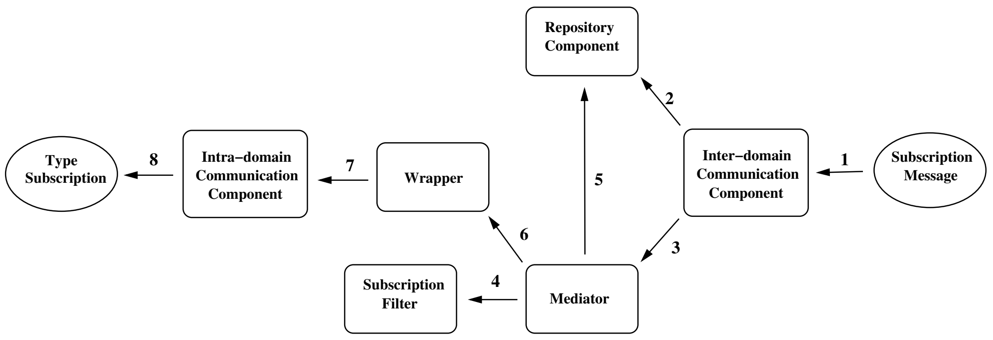


Fig. 5.8: Handling a Foreign Subscription Message

A subscription message from a foreign domain is a cross-domain subscription. When arriving at the local gateway it is a subscription for a contract type. The gateway first checks whether a contract for the contract type exists within the local domain. This check is performed with the repository component (this check should never fail because the inter-domain communication module of a gateway automatically delivers subscriptions to the correct domain). If the contract type exists in the repository, the subscription is extracted from the message and passed to the subscription filter. The mediator turns the subscription into a type subscription after retrieving the appropriate translation function from the repository. In theory, the filter is only required when the local component event system does not support the same expressiveness in its subscription language. This is very difficult to manage in a federation and it is thus a much more sensible approach to use a filter module, which provides *gateway filtering.* A gateway subscribes to an event service without any filtering expression, and applies the original filter when receiving the event instance from the component event system. The subscription filter manages subscriptions and filter expressions.

The original contract subscription is stored by the subscription filter for later use. The type subscription is passed to the wrapper component. The wrapper translates the subscription to the representation used by the component event system. The translation here depends on the component system. In section 5.6 we will show how this works with the HERALD event architecture. The wrapper then passes the translated subscription to the intra-domain communication layer. From here the subscription is passed to the appropriate component within the local domain for event subscription.

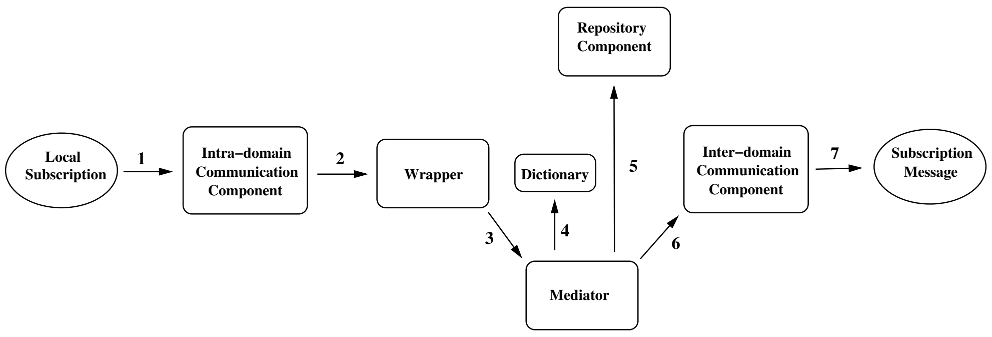


Fig. 5.9: Handling a Local Subscription

5.3.5 A Local Subscription

A local subscription message originates within the component event system. It passes through the intra-domain communication layer to the wrapper component, which translates it into a *FedReg* subscription. The dictionary is queried to determine the appropriate translation function. This is retrieved and applied to the subscription. The dictionary is then queried to determine which domains must receive the subscription, since the subscription can be for multiple domains depending on the contract type (a global or a domain-to-domain contract). The message constructor turns the subscription into a message and passes it to the inter-domain communication layer. From here it is passed to the appropriate domain(s).

5.3.6 A Foreign Publication

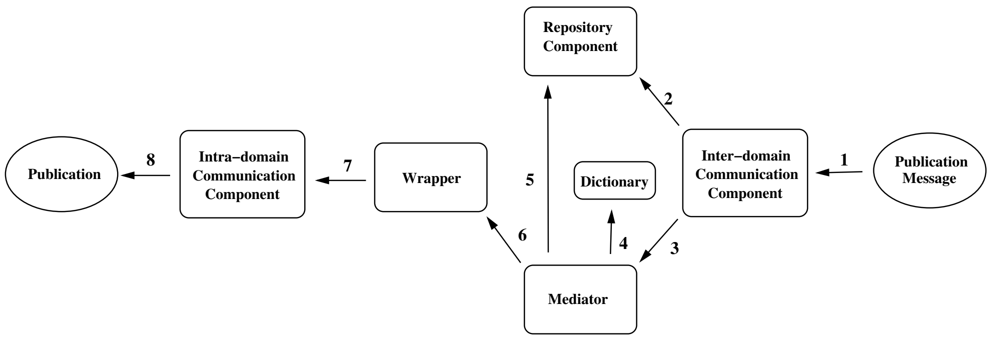


Fig. 5.10: Handling a Foreign Publication Message

The inter-domain communication layer of the local gateway receives a publication message from a foreign domain. It is validated by checking that a contract exists between the foreign domain and the local domain for that event type. The mediator uses the dictionary to apply the correct translation function to the event instance of the publication message. The message is passed to the wrapper component, which translates the event instance into the representation used by the local event architecture. From there the event instance is passed to the intra-domain communication layer, which distributes the event instance to the local clients.

5.3.7 A Local Publication

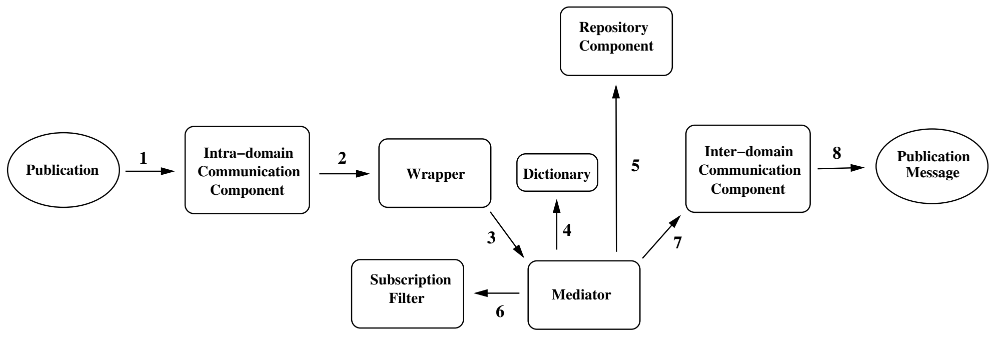


Fig. 5.11: Handling a Local Publication

A local publication message originates in the local event architecture and is passed to the gateway via the intra-domain communication layer. The wrapper component translates the event instance from the local data model to the global data model. The event instance is passed to the mediator. The mediator queries the dictionary. This returns the information necessary to retrieve the correct schema translation function(s) from the repository. The event is then translated into the contract instance(s) by the mediator. At this point the filter checks the event instance against its subscriptions. If the event instance matches a subscription, the event instance is turned into a publication message and passed to the inter-domain communication component. This component is aware of the foreign domain(s) participating in the contract and the event is disseminated to the foreign domain(s).

5.4 Inter-domain Communication

The inter-domain communication component of a gateway is based on HERMES [Pie02, PB02], a distributed event-based middleware architecture developed in the OPERA group[2] at the University of Cambridge Computer Laboratory. This component man-

[2] HERMES was developed and implemented by Peter Pietzuch.

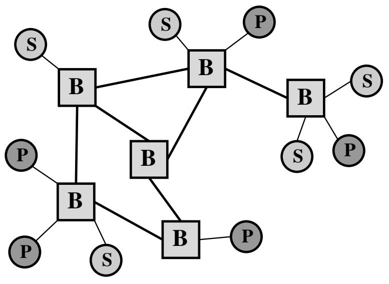


Fig. 5.12: A HERMES Application

ages the communication between gateways of domains. We describe the architecture of HERMES in this section and explain how it is used by the gateway.

5.4.1 The HERMES Architecture

The two main components in HERMES are event brokers and event clients. Clients can be either event consumers (subscribers), event producers or both. They use the middleware service provided by the brokers to communicate with each other. The brokers make up the actual middleware and provide the distribution functionality needed by the event publishers and subscribers. This allows clients to be light weight. Clients communicate with brokers to publish and subscribe (see figure 5.12). Clients connect to a broker in the network before using the middleware services. The brokers are responsible for accepting subscriptions from clients and delivering event instances to interested subscribers. Brokers are connected in the network in a random topology and communicate with their neighbor via messages. These messages either include event instances or network maintenance and configuration information. Event instances are translated into messages and routed along the network of brokers to the subscribers. Scalability is a key aspect of HERMES and thus it tries to filter the subscriptions as close as possible to the event producers.

HERMES creates special nodes called *rendezvous nodes* for advertisements and subscriptions. *Rendezvous nodes* are established when new types are added to the system. The broker whose numerical identifier lies closest to the hash value of the type name will function as a node for this event type. The nodes guarantee that a subscription finds all paths set up by previous service advertisements and then travels along the reverse path toward the event source. Since a subscription doesn't know where the sources are in the network, it is routed to the *rendezvous nodes* initially. Whenever it hits an advertisement path it follows it. When it reaches the actual *rendezvous node*, the subscription is discarded. The rendezvous node maintains an *event dissemination tree* used for delivering events to interested subscribers. *Event dissemination trees* are essentially the paths taken by subscriptions to find advertisements and the reverse paths of these. The *rendezvous node* will or will not be part of that tree.

For building these nodes and trees HERMES uses an overlay routing network. This is a logical application-level network built above a general network layer. The nodes of the overlay network are able to route messages between each other. This allows for the use of sophisticated routing algorithms because the routing is done at the application level. It is important to note that the network can dynamically adapt its topology during the lifetime of the system. This means nodes can be added or removed from a running system without causing problems to the overall functioning of the application.

Unique identifiers are associated with each node in the network. A `route(message, destination)` function is provided, which allows brokers to send messages to other brokers. The overlay network provides transparent fault-tolerance mechanisms that can handle link and broker failure. Furthermore, it handles the connection and disconnection of clients to and from the network. The overlay network also allows brokers to discover rendezvous nodes for the building of event dissemination trees, used for routing events from publishers to subscribers. The rendezvous nodes are replicated throughout the network to deal with possible network failure.

HERMES is based on the ideas developed in CEA (see chapter 2) and supports proper typing. Every event is associated with an event type. Event types are defined in a schematic manner (XML Schema) and contain a type name and set of attributes with name/value pairs. This allows for subscriptions at a finer granularity than topic-based systems. At the same time it allows for grouping of messages according to types (topics). This combination makes the system scalable and flexible. Having proper typing enables the brokers to do runtime type checking, making the system more robust.

The architecture is layered. At the lowest level is the network layer. The second layer consists of the overlay routing network for the brokers. The third layer provides the functionality of a topic-based *publish/subscribe* system through the rendezvous points that handle specific event types and manage the publication, subscription and event dissemination trees for that event type. The fourth layer is the *type and attribute-based publish/subscribe* layer. This layer distributes filter expressions throughout the network to support source side filtering. The *event-based middleware* layer is the fifth layer and provides the API used by application designers. It provides interfaces for event publication and subscription as well as adding new or removing existing event types from the system. This layer can support other middleware services such as fault-tolerance, storage, transactions, event type discovery and mobility support.

5.4.2 Extensions to HERMES

HERMES has been extended to handle other types of messages needed by the federated architecture. These messages were described in the previous section. Besides the messages, the subscription style of HERMES has been enhanced. In the original HERMES subscriptions are XPath expressions [Con02b]. These are used to check for the existence of nodes in XML objects. One difficulty with this expression language is that it can only conduct searches over a single XML object. Furthermore, it is a difficult language to translate to and from. This means it is not suitable for a federation style architecture. We have therefore extended HERMES to use the more powerful subscription language *FedReg*.

5.4.3 HERMES within the Federation

The inter-domain communication component of a gateway has the same functionality within the federation as a broker does in a HERMES application. When a domain with a gateway joins a federation, other gateways are notified about the new broker in the network. These gateways can be connected in a random topology. This topology can change at runtime when new brokers are added to the system or existing ones are removed.

The other components of a gateway act as a client does within a HERMES application. A direct link exists between the rest of the gateway and the brokering component. The communication between client and broker in a HERMES application is the same as between the inter-domain communication component and the rest of the gateway. The gateway appears as both an event producer and consumer to the broker, depending on whether it is subscribing to or delivering a contract type. The inter-domain communication component of a gateway handles the four basic message types described in section 5.3. These are used to set-up rendezvous nodes and build dissemination trees for event delivery. Furthermore, it supports the four new message types used by the federation.

```
<complexType name="Bus Location">
 <sequence>
  <element name="Line" type="xsd:string"/>
  <element name="Time" type="xsd:time"/>
  <complexType name="Location">
    <sequence>
     <element name="Street" type="xsd:string"/>
     <element name="City" type="xsd:string"/>
     <element name="Country" type="xsd:string"/>
    <sequence>
  <complexType>
 <sequence>
<complexType>
```

(a) The Contract Type

```
<Bus Location>
 <Line>7788</Line>
 <Time>13:27:23</Time>
 <Location>
   <Street>Main Street</Street>
   <City>London</City>
   <Country>London</Country>
  </Location>
 <Bus Location>
```

(b) A Contract Instance

Fig. 5.13: Event Translation I

5.5 Gateways: The Mediator Component

Every gateway in the federation shares mediator, repository and subscription filter components. The components are identical for every gateway. In this section we describe how the mediator component functions and interacts with the other components of the gateway. Particular attention is paid to the translation module. We use examples to show how translations are performed and describe the logic that manages the contract types and translation functions. The translation module has been implemented using *Xalan*, a generic XSLT processor, and *Xerces*, an XML parser.

5.5.1 Event Translation

The translation module contains an XSLT processor. This processor takes two inputs, an XML object describing the event instance or a subscription and an XSLT object containing the translation function. An event instance coming from a foreign domain arrives at the local gateway in the form of a publication message. The event instance is extracted from the message. Figure 5.13(a) shows the schema for a contract type called *Bus Location.*[3] This is an event type that shows bus locations within a city. An instance of this type is shown in figure 5.13(b). The schema document shows that the *Bus Location* type contains three attributes or elements. The elements *Line* and *Time* are of the built-in types *string* and *time*, respectively. The prefix `xsd` indicates

[3] For reasons of clarity we only show the type definition, leaving out other XML data associated with an XML Schema document. We do the same for XML documents describing the event instances.

```
<complexType name="Bus Position">
 <sequence>
  <element name="BusLine" type="xsd:string"/>
  <element name="Time" type="xsd:string"/>
  <element name="LocationName" type="xsd:string"/>
 <sequence>
<complexType>
```

(a) The Local Type

```
<Bus Position>
 <BusLine>7788</BusLine>
 <Time>1:27 pm</Time>
 <LocationName>Main Street,
       London</LocationName>
<Bus Position>
```

(b) A Local Event Instance

Fig. 5.14: Event Translation II

the namespace for these types. The *Location* type is a complex type consisting of three sub-elements. These elements are *Street*, *City*, and *Country*. They are all of the simple built-in type *string*.

When the contract type is first received by the gateway it is validated by checking it against the contract schema stored in the repository. The publication message contains additional information which can be checked to validate the contract type. This is information about the origin of the event. Information about contract partners is also maintained by the gateway and every contract type can be checked against this information. These two checks provide security for a domain and are not necessary, depending on the trust level within the federation. Once validated, the appropriate translation function is applied to the contract type.

Figure 5.14(a) shows the schema for the corresponding local type in the example. The type *Bus Position* contains three simple elements in its definition. These are *BusLine*, *LocationName* and *Time* and are all of the local built-in type *string*. Comparing this schema with the one of figure 5.1(b) highlights some of the conflicts that the translation function needs to resolve. Firstly, the two event types have a different name. The name of the contract type is different from the name of the local type. Secondly, the elements *Line* and *BusLine* are of the same type, but have a different label. They are likely to be similar, but the label needs to be translated. Thirdly, the elements *Location* and *LocationName* are similar, but are again of a different type. One is a complex type with three sub-elements while the other is of the built-in type *string*. Also, their labels are different. Fourthly, the *Time* element in the two schemas is of a different type. In the contract schema it is of the built-in type *time*, whereas in the local schema it is of type *string*. These differences are all results of structural semantic heterogeneity and can be resolved. Figure 5.15 shows an XSLT document with the translation function to convert from the contract type to the local type.

```
<xsl:template match="Bus Location">
  <Bus Position>
    <BusLine><xsl:value-of select="Line"/></BusLine>
    <Time><xsl:value-of select="extend:time_conv(time(Time))"/></Time>
    <LocationName><xsl:value-of select="Street"/> <xsl:text>, </xsl:text>
    <xsl:value-of select="City"/></LocationName>
  </Bus Position>
</xsl:template>
```

Fig. 5.15: An Event Translation Function

Three things are of interest in the translation function. Firstly, the example illustrates how to handle naming conflicts. This is done by the conversion of the element *Line* in the contract type to *BusLine* in the local type. Secondly, a type conversion from a complex type to a simple string type is shown. The *Location* element with its three sub-elements is concatenated into the *LocationName* element. One of the sub-elements, *Country* was ignored by the translation function. Finally, the example shows XSLT extensibility. This refers to syntax that allows an XSLT function to reference external objects during document processing. It allows documents to be transformed using information from an external reference source or function declaration. The function is not defined in the example. It is a simple function that takes an element of type *time* and converts it into a *string* type. The functions must be provided as part of the contract.

There are various ways in which extensibility can be used to help resolve structural semantic heterogeneity:

- An extension function can perform a database lookup in a table to resolve domain conflicts. For example, a lookup function can determine that *pp202* in domain *A* is *peter* in domain *B*.
- Some transformations require a calculation on a data item in the input document. For example a temperature measurement can be converted between Celsius and Fahrenheit or new values can be calculated from existing ones.
- Sometimes new information needs to be added during the transformation process. An example of this is a timestamp value that can be inserted into the output document.

5.5.2 Subscription Translation

Subscription translation is similar in principle to event translation, but there are added difficulties caused by the varying levels of expressiveness provided by the subscription mechanisms of the component event systems. We call this *subscription heterogeneity*. This heterogeneity is partially handled by the filter component and partially by the mediator component of a gateway. Subscription translation only occurs in the domain where the subscription originated. Here a local subscription is translated into a contract subscription before being passed on to the foreign domain. In the foreign domain the contract subscription is translated into a local *type subscription*. This local type subscription is passed to the component event system. Event instances of that type are passed from the component event system back to the local gateway and translated into contract types. The filter module then checks the contract instance against the contract subscription. We call this mechanism *gateway filtering*. Gateway filtering means event instances are filtered at the gateway rather than within the component event system. We have opted for this approach for various reasons.

There are instances when the subscription language of one domain is more expressive than the subscription language of another domain. When translating a more expressive subscription to a less expressive subscription, information is lost and the subscription will cause events to be filtered at a lesser granularity than intended. This means filtering needs to be done at the point where the loss of information for a subscription translation takes place. This loss can only occur at the foreign gateway because we are assuming that the federation subscription language can represent any component event system's subscription language. If that was not the case, filtering would have to be done at the gateway of the domain where the subscription originated.

By choosing a very expressive subscription language and in trying to provide *source-side filtering* or *gateway filtering*, we have made translating subscription messages more difficult. Figure 5.16 highlights an instance where this is the case. The subscription shown in figure 5.16(a) is a contract subscription. Translating this into a subscription of the local type *Bus Position* is impossible due to the concatenation of the complex *Location* attribute into the simple *LocationName* attribute.

We have made a trade-off in our architecture to handle this type of heterogeneity. We have opted not to allow *complex subscriptions* for those elements where concatenation into string types or its inverse occurs. By this we mean that subscriptions that are part *wildcard* and part value for complex structured attributes, as is the case in the example in figure 5.16, are not permitted. This check is performed by the gateway where the subscription originates. In doing so, we slightly reduce the expressiveness of the language, but are still able to provide *gateway filtering*. In instances when the concatenation occurs in the mediator component of the foreign gateway, the subscription restriction does not apply. We have thus only reduced the expressibility of the subscrip-

```
<xce:Subscription>
 <xce:Template type="Bus Location">
  <Line>7788</Line>
  <Time>*</Time>
  <Location>
   <Street>Main Street</Street>
   <City>*</City>
   <Country>*</Country>
  </Location>
 </xce:Template>
</xce:Subscription>
```

(a) Foreign Subscription

```
<xce:Subscription>
 <xce:Template type="Bus Position">
  <BusLine>7788</BusLine>
  <Time>*</Time>
  <LocationName>Main Street, *</LocationName>
 </xce:Template>
</xce:Subscription>
```

(b) Local Subscription

Fig. 5.16: The Subscription Translation Problem

tion language in instances where the concatenation occurs in the mediator component of the local gateway.

One possible way to resolve this problem would have been to move the filter component to the gateway of origin. In such an architecture, the subscription from the domain of origin is translated into a type subscription, as we currently do in the foreign domain, before being sent to the intended domain. This means all event instances of a particular type are sent across the domain boundary and filtering occurs at the local domain (from the point of view of the subscription). This is less scalable and still does not completely resolve the problem of *subscription heterogeneity.* The problem can only be resolved by extending the federation subscription language to handle complex string manipulations.

From a review of event architectures and messaging systems, this problem will not be encountered in practice since complex subscription mechanisms are generally not supported by component event architectures. All architectures with topic-based subscription mechanisms will not be faced with this problem. The same is true for component event or messaging systems where elements are non-complex. Other forms of structural semantic heterogeneity in subscriptions can be resolved by the architecture, including domain, naming, and type conflicts.

5.6 Gateways: The Wrapper Component

In this section we describe an implementation of a wrapper component for a specific event architecture. We do this for a HERALD application. We focus especially on how the wrapper handles type and subscription translations. Based on this example, we develop a common approach and define interfaces for wrapper and mediator components. This allows wrapper components to be developed separately and treated as a black box.

As previously mentioned, the implementation of a wrapper component depends solely on the underlying event architecture. In a federation where the component event systems and the federation DDL and subscription language are the same, this module is not required since there would be no data model mismatch. In such instances only the mediator component is necessary.

A wrapper component maps event publications, subscriptions and definitions between data models. While event instances are translated between the two data models, subscriptions are only translated in the domain of origin and partially translated in the destination domain. A wrapper in the destination domain only translates a *type subscription* rather than a fine-grained *attribute subscription.* Wrappers can help overcome semantic heterogeneity by extending event definitions to include descriptive information for the event types published by a component event system. This can be done by adding *comment labels* to XML documents for certain attributes. This is not a requirement since descriptions themselves can cause semantic uncertainty and tend to require human input. But in many instances it can be useful. For example, wrapper components can include simple comment elements in their type definitions explaining what currency or which unit of measurement is being used for an attribute when it is unclear.

5.6.1 A HERALD Wrapper

In HERALD new event types can be created and added to a running system. New types are created by specifying an event type name and a list of attributes. Each type has a unique identifier. HERALD is implemented in Java and thus supports all the normal built-in Java types for event creation. The system guarantees uniqueness of types within the system. Interfaces for querying event schemas and for event registration are provided for event clients. The wrapper component makes use of these interfaces to communicate with the component event system.

From the perspective of the component event system, the gateway appears either as an event producer or an event consumer. The event client and event source subcomponents of the intra-domain communication layer communicate with the HERALD event system, while the federation interface communicates with the rest of the gateway (see figure 5.17). A translation module sits between these components and handles the

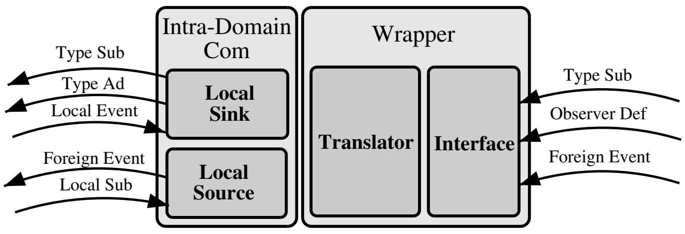


Fig. 5.17: The HERALD Wrapper

translation between the local and global data model. It does this for event instances, subscriptions and event schemas. Event instances are translated in both directions, translating between the two data models. Similarly, subscriptions are translated bi-directionally, but a cross-domain subscription in the destination domain is only a type subscription, making the translation simple. The HERALD wrapper translates local subscriptions into *FedReg* subscriptions and simple *FedReg* type subscriptions into HERALD type subscriptions.

An event translation uses a simple algorithm for events coming from the HERALD domain. Event instances contain attribute/value pairs and type information. This is used to construct the XML object. In the other direction, the XML object is traversed and the nodes containing label/value pairs are used to construct the event instance in the HERALD data model.

Subscriptions in HERALD are templates. They are instantiated by providing the schema of the event type for which a template is created. Attributes can then be filled in or left blank, thus providing attribute based subscriptions. An example for a login event is shown below. This event type monitors users logging into systems on a network. In the example the first and second attributes are given values. If there are other attributes that are left blank, they count as wildcard values. The event client uses this template to subscribe to an event service.

```
loginTemplate = new EventTemplate(loginEventSchema);
loginTemplate.setString(0, "pp202");
loginTemplate.setString(1, "tristan.cl.cam.ac.uk");
```

Since subscriptions from a foreign domain are only type subscriptions, in the above example an empty template would have been used to register interest in the login event. All that is needed is the type name. This is used to retrieve the schema of an event type to create the empty template. The translation from the global data model to the local data model for a subscription is then straight forward.

The reverse operation, transforming a local subscription into a *FedReg* subscription, is very similar. The intra-domain communication layer receives a local subscription and uses the template object to create a *FedReg* object using the labels and values from the template of the subscription.

All event types are published federation wide by the gateway. The intra-domain communication component (the source sub-component) connects to HERALD's event broker, which manages event definitions for the domain. It retrieves the schema information for every event type. HERALD event definitions are translated into XML Schema objects by the translation module. Again, the type definitions in HERALD are objects that contain label/value pairs that are used to create nodes in an XML object. The translated definitions are passed to the gateway where they are stored in the repository.

Originally, when new event sources were introduced to HERALD, the wrapper module was not informed and thus did not automatically send the new event type information to the gateway for storage in the repository. HERALD was extended to handle dynamic changes to the system. The wrapper periodically queries the broker to check whether new services have been added. This has proved helpful since event sources may also be removed from a HERALD application at runtime. In those instances, the wrapper module also sends updated information to the gateway so that it can correctly update the repository.

HERALD clients handle cross-domain subscriptions for observer types originating in a foreign domain by querying the local event broker. The broker provides the binding for the available services. Since the gateway appears as a HERALD event source, clients simply register interest for an event type with the event source sub-component of the intra-domain communication component. The wrapper handles the subscription translation, passing the translated subscription to the mediator component.

Observer event types are introduced into a HERALD application through the intra-domain communication sub-component. When the local gateway receives an observer message with a type definition and a translation function, the observer type definition is translated and passed to the source sub-component. This component registers the new event type with the HERALD event broker. Local event sinks can then dynamically discover the new service when querying the event broker and register with it.

5.6.2 Wrapper and Mediator Interfaces

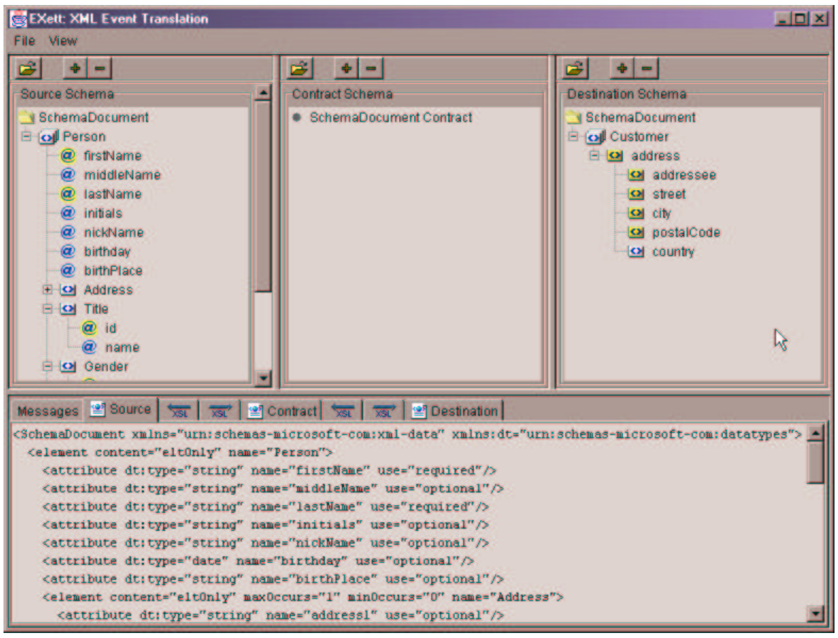
We can now define common interfaces for both the wrapper and mediator component of a gateway. In doing so we abstract away from the detail and leave the implementation of the wrapper to the application designer. By providing the interfaces to the application designer, gateways can be constructed more quickly and easily.

The wrapper module requires three interfaces, which the other parts of the gateway use for communication. These are used for observer type publications, subscriptions and notifications. The type publication interface takes an observer type definition as an argument. This is a string which describes an XML Schema object. The subscription interface takes a string as an argument. The string contains a *FedReg* object with the subscription information. The other interface is for event notification. Again, this interface takes a string object as an argument which is an XML object describing an event instance. The mediator module requires three interfaces, which the wrapper uses to communicate with the rest of the gateway. These are for event notification, subscription and event type publication. All three interfaces take a string as an argument. For the subscription interface this is a *FedReg* object, for the event notification interface an XML object describing an event instance and for an event type publication an XML Schema object with an event type definition.

5.7 Contract Creation

Semantic heterogeneity between systems is resolved during the process of contract creation. We have created a tool with a graphical user interface for constructing contract types and translation functions (see figure 5.18). The tool sends a *discover message* to all domains participating in the federation. It receives a *discover reply message* from all domains. These messages are used to display the available event types of the federation, including contract types.

The tool allows users to create contract types and translation functions via drag-and-drop. In instances where the application is uncertain the user is prompted to resolve a particular conflict. Once a new contract type and the translation function(s) are created, the tool generates a *type* and *contract message* and sends it to the participating gateways. When a new contract is created between an existing contract type and a new type, *contract messages* are sent to all domains involved so that these can either store the translation function or can update the dictionary with the new contract information. The message for the gateway from which the contract type originated is



```
<SchemaDocument xmlns="urn:schemas-microsoft-com:xml-data" xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <element content="eltOnly" name="Person">
    <attribute dt:type="string" name="firstName" use="required"/>
    <attribute dt:type="string" name="middleName" use="optional"/>
    <attribute dt:type="string" name="lastName" use="required"/>
    <attribute dt:type="string" name="initials" use="optional"/>
    <attribute dt:type="string" name="nickName" use="optional"/>
    <attribute dt:type="date" name="birthday" use="optional"/>
    <attribute dt:type="string" name="birthPlace" use="optional"/>
    <element content="eltOnly" maxOccurs="1" minOccurs="0" name="Address">
      <attribute dt:type="string" name="address1" use="optional"/>
```

Fig. 5.18: Contract Creation Tool

a simple message containing information about the new domain joining the contract. The new domain receives a normal *contract message* with information about contract partners and the translation function.

The tool handles certain requirements that were described in chapter 4. For example, it does not allow contracts between two contract types, or contracts between two types from the same domain. Furthermore, it checks newly created contract types since duplicate contract types are not permitted.

5.8 *Observer Functions*

Observer functions are created with the same tool as contracts. Rather than displaying all event types available within the federation, the tool, when in observer function mode, only displays contract types and for each contract type a list of domains taking part in that contract. Via drag-and-drop the user is able to create an observer type. The appropriate translation function is generated automatically (the user is again prompted when there are conflicts). The process is exactly the same as for contract creation.

Once an observer type and an observer function have been created for a particular contract type, the tool generates an observer message to inform the domain about the new type. In instances where the observer type is identical to the contract type, a message with an empty function element is generated.

5.9 Summary

This chapter described an architecture for gateways, the main building blocks in an event federation. We have used basic tools to achieve interoperability and flexibility with this architecture. These tools included XML Schema for event definitions, XML for event representation and subscriptions, XSLT for representing translation functions, *Xerces* for XML parsing, *Xalan* for XML transformation, *Xindice* for XML storage and HERMES for content-based message passing. A gateway component is responsible for communication between entities of a federation and translation of XML objects. The design of the gateway architecture was based on the event federation paradigm developed in the previous chapter.

We described the main components of a gateway. These are the inter-domain and intra-domain communication layers, the wrapper, the mediator and the subscription filter. The mediator component resolves structural semantic heterogeneity using the translation functions of the contract and applying them to event instances, both local and foreign. The mediator uses a dictionary to determine which function to apply to which event instance. After translation, a message constructor turns local events into publication messages and lets the inter-domain communication layer deliver it to the appropriate domains.

We described HERMES, a content-based messaging middleware which we use to connect different domains. HERMES is scalable, using intelligent routing algorithms by setting up *rendezvous nodes* which create and manage *event dissemination trees*. HERMES codes its messages in XML. It supports *type*, *advertisement*, *subscription* and *publication* messages. HERMES has been extended to also support *contract*, *discover*, *discover reply* and *observer* messages. These messages are used for communication between domains of the federation. HERMES' subscription language has been changed from XPath to *FedReg*. *FedReg* is a pattern-matching language based on XML. It includes a non-XML filtering language for specification of more complex constraints, using logical predicates.

We showed how event and subscription translation is handled by the mediator component. Event translation occurs bi-directionally. The mediator uses the dictionary to determine which translation function to apply to which event instance. Event instances may need to be translated into multiple representations, depending on the number of contracts for that event type. An event instance from the local domain is translated into contract types for each contract that exists for that type. A contract event from a foreign domain is translated into an observer type.

We have made a trade-off in our architecture. We translate attribute-based subscriptions into type subscriptions at the destination gateway and let the subscription filter module of the gateway provide *gateway filtering.* This was necessary because complex subscription languages cannot be translated into simpler ones without information loss unless the filtering is done where the translation occurs. *Gateway filtering* was also necessary to partially overcome *subscription heterogeneity.* This is encountered when subscription in event types with structured attributes need to be concatenated into a single attribute when translated. In those instances the filtering mechanism in the component event systems would need to support sub-string filtering facilities to be able to handle these subscriptions.

We briefly described the requirements of the wrapper component and discussed a specific implementation for HERALD. Using this as a basis, we defined standard interfaces for the wrapper and mediator components.

Finally, we discussed a tool for contract creation. The tool automatically generates translation functions for a contract. The tool queries all domains in a federation for the available event types and has a graphical interface to create contracts using existing types. Upon completion, the contract information is packaged into contract message and sent to the appropriate domains. The same tool is also used to create observer types and observer functions. With these a domain can observe contract types locally.

6. EXPERIMENTS

This chapter describes two applications linking heterogeneous event systems. These applications are based on the federated event architecture described in the previous chapter. The first application links *Active Houses.* An *Active House* is a house where appliances and other devices are linked to a network using an event architecture. The state of the different appliances and devices is communicated to interested entities using events. In the experiment, we link heterogeneous houses, *Active Houses* with different underlying event architectures, to build an *Active Street.* We then continue this idea and provide city wide services, building an *Active City.*

We begin by describing a single *Active House* and its appliances, giving an overview of the event architecture used and the event types supported by the house. We then extend the architecture with a gateway to enable it to join a street of *Active Houses.* The other houses on the street use the same event architecture or HERALD. We describe a house based on HERALD and the event types it supports. We describe some of the contracts that exist in a street and show how events and subscriptions are translated between houses. Using the same ideas, we then describe an *Active City.* An *Active City* is constructed in one of two ways. It can be built from individual *Active Houses*, just like an *Active Street* or it can be built by federating *Active Streets.* The second example is interesting because it illustrates how the federation paradigm can be used to build federations of federations. We explore how a combination of these two designs can coexist in a federation.

Finally, we show how the event federation paradigm can be used to link heterogeneous mobile entities and connect them to services within the *Active City.* The scenario incorporates mobile devices in the *Active City* application. We show how federation using gateways is a paradigm well suited for this. Mobile devices are connected and reconnected from the network in a random fashion. When disconnected, messages for a mobile device must be stored at a gateway until the device reconnects. Devices communicate using different message formats. Translation between formats is necessary in order for heterogeneous devices to communicate. Contracts between devices can describe a common communication format and the translation functions to and from that format.

6.1 The Active Street

The first application we describe is based on our work on the *Active House*. The original *Active House* was built using a generic CEA-style event notification system to link modern appliances and devices in a house. Based on this idea, we implemented multiple *Active Houses*, using the generic and the HERALD event architectures. We linked these houses using the event federation architecture to form an *Active Street*.

6.1.1 The First House

Being CEA-based, the event architecture of the *Active House* is based on the *publish/register/notify* paradigm for communication. In the original *Active House* application, an event is a parametrized, asynchronous message which is an instance of an event type. Event types are defined through their attributes. Three different attribute types are supported, *strings*, *chars* and *integers*. Event sources publish the event types they produce. Event sinks register interest in these events, possibly providing parameter values to be matched. When an event occurs, all registered parties with matching parameters are notified.

The *Active House* demonstration [HM98] uses events to link a range of automated appliances within a virtual house. The appliances can both produce and receive events, meaning they can act as an event source and sink at the same time. They therefore have the ability to publish, register, notify and receive events as described above. On receiving an event a device can be programmed to perform some action.

The publication of the event interfaces in the *Active House* is done using an advertising agency. Event sources advertise and de-advertise their interfaces with this agency.

```
advertise(source, eventTypeName, eventTypeDef)
de_advertise(source, eventTypeName)
```

The `source` argument uniquely identifies the event source on the network. The `eventTypeName` argument gives the name of the event. The `eventTypeDef` argument gives the definition of the event type, including its attributes. This is done using a derivative of

ODL [CBB+99]. Once published, clients can query the advertising agency and register interest with an event source through the service.

As mentioned, devices in the *Active House* can both produce and receive events. For example, the *stereo system* may register interest with the event source *doorbell*. When the event *doorbellRing* occurs the stereo system is notified. The stereo system may also act as an event source, which can notify other devices on the network of events, such as a *volumeChange* event.

When receiving an event, a device typically performs some action. The *Active House* has a rule system, whereby the owner of the house can specify what action an event sink takes when it is informed of a particular event occurrence. This is similar to *event/condition/action* rules [MD89] used in active systems.

6.1.2 The Devices and Occupants

In the original demonstration, the house consists of six rooms. Rooms are connected via doors. The house is occupied by a family of five. Family members move through the house in a random fashion. The house contains the networked devices listed below.

Security Cameras Each room is equipped with a security camera. These cameras continuously monitor the rooms by taking snapshots of the people in the room (approximately every second). The six cameras are connected to a security system. This extends the functionality of a single security camera. By grouping the cameras into a single component, one can receive information concerning the whereabouts of people from a single device. The security system produces two event types, *enter* and *leave* events. They are both parameterized, with each event type having the same two parameters, a `roomName` and a `personName`.

Toaster The kitchen contains a toaster. It produces *foodReady* events when toast has been toasted. The event has one attribute. It performs a simple action, which starts the toasting.

Doorbell The doorbell is similar to the toaster device in that it only produces one event, without any attributes, and can only perform one action. The event it produces is a *doorbell* event and the action is a ring.

Stereo System The lounge has a stereo. It can produce two events, *mute* and *un-mute*. Besides these two events, the stereo can also be switched on and off.

Lights Each room has a light. These lights produce *light* events that are parameterized.

The parameters indicate whether the light is turned on or off. The parameter is of type *integer*. The light can also be set on a timer, in which case a light will switch itself off after a certain time once everyone has left the room.

Coffee/Tea Machine The kitchen has an intelligent machine which can make tea or coffee. It sends a *foodReady* event whenever tea or coffee has been made. The *foodReady* events are parameterized because we have a *foodReady* event for tea, coffee, and toast. The machine can perform two actions, `makeTea` and `makeCoffee`.

Monitor The monitor device allows the owner of the house to watch for specific event occurrences, by outputting them to a screen. Just like the other devices, the monitor registers interest with specific event sources. When an event occurs, the monitor is notified and the event information is displayed on the screen.

These are the main event sources and sinks in the house. They are linked and able to communicate with each other via the event architecture. The fact that devices can both produce and receive events is convenient for the event federation architecture, since the gateway for the *Active House* will act both as an event producer and consumer. It is a consumer of local events on behalf of the federation and a producer on behalf of the local domain for contract events with observer functions.

6.1.3 The Controls

Each device has an interface that can be used to control it. The interfaces are used by each device's control panel. This allows users to interactively control and manage the behavior of a device. Five functions can be controlled via this interface. Below we describe the interfaces and the functionality of the controls.

Activation/Deactivation A device can be activated or deactivated. This means that the user of the house can control which devices are available on the network. Furthermore, if a device can perform actions they can be manually started through this interface. For example, the toaster can be made to start toasting.

Registration This interface is used to register interest with event sources on behalf of the device. An event source and a set of parameters can be specified. This allows for attribute-based registration. For example, an *enter* event can include a `name` and `room` argument. These parameters can also be left blank to indicate a wildcard value.

Source List This lists the event sources with which the current device has registered.

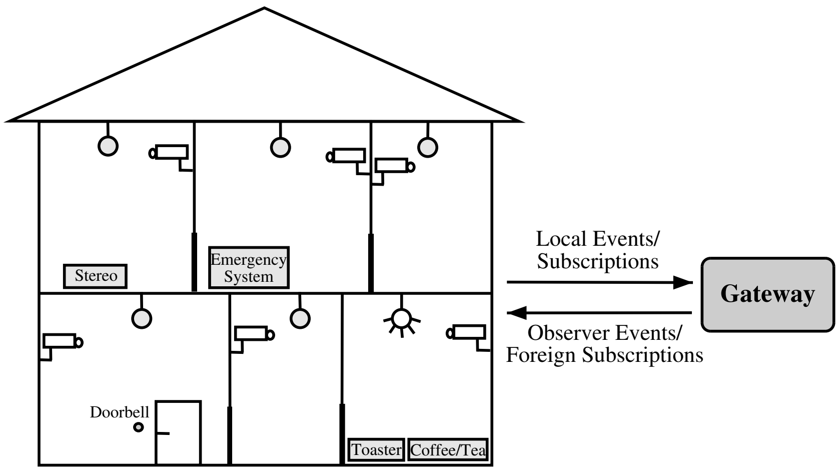

Stereo
Emergency System
Local Events/ Subscriptions
Gateway
Observer Events/ Foreign Subscriptions
Doorbell
Toaster
Coffee/Tea

Fig. 6.1: An *Active House*

The list also shows the parameters for each registration. The user can unregister with event sources through this interface.

Sink List This lists all clients on the network that have registered interest with the current device. The list also details event parameters on which the current device will notify.

Rules The user can specify actions to be taken when specific events have occurred. By specifying an event, a condition and an action, the user of the house can control the devices. The *event/action/condition* rule `enter(*, Kitchen) -> makeToast` will cause the toaster to make toast whenever the *enter* event occurs with the condition that someone has entered the kitchen.

6.1.4 Extending the Active House

The *Active House* demonstration was developed without event federation in mind. The event types are very simple and we have therefore extended it with another event source. This source is an emergency notification system, which produces three types of events. The first is a *fire* event and has three parameters, a street name, a house number, and a city name. The second event is a *break-in* event and has four parameters, a street name, a house number, a city name and a room where the break-in was detected. The final event is a *power/gas* event and has five parameters, a street name, a house number, a city name, a problem description and an urgency value.

The fact that the original *Active House* was developed prior to developing and implementing the federation architecture is important because it shows that the paradigm is able to integrate existing applications of heterogeneous event architectures. The house is integrated into the federation by extending it with a gateway component. This component is based on the architecture described in the previous chapter. The gateway of the house translates local event instances into contract events, foreign subscriptions into local subscriptions and local subscriptions into contract subscriptions.

In the previous chapter we illustrated how a wrapper module and the intra-domain communication component are unique to each domain, while the mediator, subscription filter and inter-domain communication module are identical for each gateway. The original house was therefore extended with a wrapper module for the generic CEA-based event system.

Event definitions are retrieved from the advertising agency. The definitions are translated into the global data definition language used by the federation, XML Schema. A type definition translation performed by the wrapper for an *enter* event is shown in figure 6.2. The gateway publishes all event types the house is willing to share with the federation. For the purpose of this application, all events generated by the house were published, except those generated by the stereo system, toaster and monitor.

```
class enter extends event
{
  attribute string personName;
  attribute string roomName;
};
```

(a) A Local Event Definition

```
<complexType name="enter">
 <extension base="baseEvent">
  <sequence>
   <element name="personName" type="xsd:string"/>
   <element name="roomName" type="xsd:string"/>
  </sequence>
 </extension>
</complexType>
```

(b) A Global Event Definition

Fig. 6.2: Data Definition Translation

The wrapper performs a similar translation for event instances. In the *Active House*, events are instances of *Java* classes and are therefore easily translated into XML objects. The reverse translation for observer objects was not possible because in the original *Active House* new event types could not be added to a running system. For this reason the wrapper component did not need to translate subscriptions for the original *Active House*.

6.1.5 A Second House

We implemented a second *Active House*, using the HERALD event architecture. Unlike the original house, this event architecture allowed us to add new event types to a running system. This meant that observer functions could be added to the house for observing event instances of the contract types within the house. In the HERALD house only the monitor device was implemented as an event source and event sink. Other devices were only implemented as sources. The devices and their event types are listed below:

Security Cameras These devices are again grouped into a single security system device which monitors all the rooms in the house. One can receive information concerning the whereabouts of the people from this device. Again, the event produces *enter* and *leave* events. The events have attributes `room`, `person` and `houseID`. The first two are of type string, while the last is an integer.

Doorbell This device produces a *ring* event. The device is also capable of identifying the person ringing the doorbell by doing a real-time fingerprint analysis when the person rings the doorbell. The event therefore has an attribute `name` of type string.

Coffee/Tea Machine The coffee/tea machine produces a single event. This *drinkReady* event has an attribute *type* which is a string and comes from the domain of values tea, coffee or both.

Emergency System This service produces two types of events. The first is a *fire* event and has three parameters, a street name, a house number and a city name. All parameters are of type string. The second event is a *break-in* event, which again has three parameters, all of type string. These are a street name and number, a city name and a room where the break-in was detected.

Monitor This device is the same as in the original active house. It allows the owner of the house to watch for specific event occurrences. The monitor registers interest with event sources and when an event occurs, the current output device is notified and displays or logs the information.

The HERALD house was extended with a gateway component in the same manner as the original *Active House*. The details of this wrapper component are described in the previous chapter in section 5.6.

6.1.6 The Street

The houses are not part of the federation until they have formed/joined a contract or provided an observer function for a contract type. In this application we have created two global contracts and established two domain-to-domain contracts. The global contracts concern the events produced by the emergency systems of each house. The first contract defines a global *fire* event, the second a global *break-in* event. The *fire* contract may have been defined by the fire department of a city, while the *break-in* contract may have been defined by a police service to implement a *neighborhood watch* program. Each house wishing to use the services provided by the global contract types must join the contracts. The schemas for both of these contract types are shown in figure 6.3.

```
<complexType name="fireEvent">
 <extension base="baseEvent">
  <sequence>
   <element name="street" type="xsd:string"/>
   <element name="city" type="xsd:string"/>
  </sequence>
 </extension>
</complexType>
```

(a) A Global *fire* Event

```
<complexType name="breakInEvent">
 <extension base="baseEvent">
  <sequence>
   <element name="houseNum" type="xsd:integer"/>
   <element name="street" type="xsd:string"/>
   <element name="room" type="xsd:string"/>
  </sequence>
 </extension>
</complexType>
```

(b) A Global *break-in* Event

Fig. 6.3: Global Contracts

The *neighborhood watch* scheme can be used by *Active Houses* in the street in two ways. Firstly, each house wishing to join the scheme can provide a translation function for the contract type, thus joining the contract and taking a passive part in the scheme. Secondly, a house can provide an observer function, allowing it to monitor the street for *break-ins* in other *Active Houses*, thus taking on an active role. Using the *neighborhood watch* scheme, *Mr Active* can go on a worry-free vacation while *Mr Herald* promises to monitor his neighbor's house. For this to happen, *Mr Active* joins the contract by providing a translation function from its local *break-in* type to the contract type and *Mr Herald* provides an observer function for the contract type. *Mr Herald* then registers interest in the observer type. He can register for all *break-in* events by specifying an empty subscription (see figure 6.4(a)) or he can just subscribe to *break-in* events from *Mr Active's* house by specifying the `houseNum` and `street` attributes (see figure 6.4(b)).

Besides global contracts the two houses can also establish domain-to-domain contracts. The first of these contracts is for a *doorbellRing* event. This contract may have been

```
<xce:Subscription>
 <xce:Template type="breakInEvent">
   <houseNum>*</houseNum>
   <street>*</street>
   <room>*<room>
  </xce:Template>
 </xce:Subscription>
```

(a) General Subscription

```
<xce:Subscription>
 <xce:Template type="breakInEvent">
   <houseNum>25</houseNum>
   <street>South Street</street>
   <room>*<room>
  </xce:Template>
 </xce:Subscription>
```

(b) Specific Subscription

Fig. 6.4: Subscriptions for the Neighborhood Watch Scheme

set-up by neighbors who often visit each other and want to ensure that when not in their house, they are still able to monitor when someone rings their own doorbell. The monitor device in the HERALD house can be used to display *doorbellRing* events from the original *Active House*. To do this, the HERALD house provides an observer function for the contract event. The other domain-to-domain contract is for an *enterRoom* event. This event may be used to notify the inhabitants of the HERALD house that someone has arrived in the *Active House*. The contract types for these events are shown in figure 6.5.

```
<complexType name="ringEvent">
 <extension base="baseEvent"/>
</complexType>
```

(a) A Global *ring* Event

```
<complexType name="enterEvent">
 <extension base="baseEvent">
  <sequence>
   <element name="room" type="xsd:string"/>
   <element name="person" type="xsd:string"/>
  </sequence>
 </extension>
</complexType>
```

(b) A Global *enter* Event

Fig. 6.5: Domain-to-Domain Contracts

By creating a contract the houses have joined the federation. With the domain-to-domain contracts the translation functions are generated when the contract type is created. A type message is sent to each of the two gateways, followed by a contract message containing the translation functions. The translation functions for the contract types must be provided by the individual domains of the federation. This happens when a house joins a global contract. A type message is created and sent to the gateway of the domain joining the contract. In the above example, the HERALD house joined the global `fireEvent` contract by creating a translation function from the local *fire* event to the contract event. This function is sent via a contract message to the gateway of the HERALD house. The same is done for the original *Active House*. We described translation functions and contract messages in detail in section 5.5.

For the HERALD house to be able to observe an instance of a contract type locally, it must provide an observer function for that contract. This was done for each of the four contract types. The observer function for the global `ringEvent` type is an empty function. This means that the mediator component of the gateway does not need to perform a translation, only the wrapper. The observer function for the `enterEvent` type is the inverse of a faithful schema translation function. This means that the observer type is the same as the local *enter* event type. The observer functions for the `fireEvent` type and `breakInEvent` are similar. In figure 6.6 we show the observer type and the corresponding observer function for the global `fireEvent` type.

```
<complexType name="neighborFireEvent">
 <extension base="baseEvent"/>
  <element name="house" type="xsd:string"/>
 </extension>
</complexType>
```

(a) An Observer Type

```
<xsl:template match="fireEvent">
 <Event type="neighborFireEvent">
  <xsl:value-of select="street"/> <, >
  <xsl:value-of select="city"/>
 <Event>
</xsl:template>
```

(b) An Observer Function

Fig. 6.6: Observer Type and Function

By providing the observer functions, the gateway of the HERALD house must locally advertise the new event types that have now been made available. It can do this because within the domain it is seen as both an event producer and an event consumer. The intra-domain communication layer informs the local event service of the new event instance available within the house. Clients can then register for observer types in the same way as they would for local event types. In the scenario where *Ms Active* visits *Ms Herald* for an afternoon coffee, but wants to ensure that she doesn't miss the plumber, she can use the monitoring service in the HERALD house to register interest with the observer type `ringEvent`. When this event occurs at her home, she is informed via the monitoring device in the HERALD house.

Figure 6.7 shows how we have extended this idea and created a street of *Active Houses*. By creating multiple instances of the HERALD and *Active House*, new houses were added to the street. This process was the same as for the original HERALD house that joined the federation. Each new instance becomes part of the street federation by joining an existing contract, like the *neighborhood watch* scheme, by creating a new contract with another *Active House* or by providing an observer function for a contract

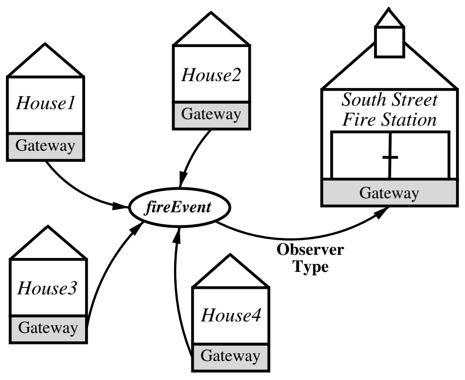


Fig. 6.7: An Active Street

type.

In the example in figure 6.7, we have also added a fire station to the *South Street* federation. This fire station has provided an observer function for the global contract type `fireEvent`. Houses wishing to join the fire service have to join the contract by providing a translation function from a local fire event to the contract type (if they have such an event). The fire station can register interest with the observer type and thus monitor the street for fires via the federated event system. The fire station was implemented using HERALD in this application.

6.2 The Active City

The *Active City* is a simulation that links *Active Streets* and other active entities in a city using the federated event architecture. The application illustrates how federations (Active Streets) can be joined to create a larger federation. The federations are extended with a gateway component which maps event types, event instances and subscriptions. Once extended, the federation has exactly the same characteristics as a domain and can therefore be easily integrated into the new federation. The only requirement is that there are no conflicts between domain identifiers of the new federation and the sub-federations. This is managed by the federation administrator.

6.2.1 The Application

Our *Active City* is made-up of three federations and three domains. The federations are the *Active Streets*, South Street, Durbin Lane and Village Avenue. The streets are built using both the generic CEA-based event architecture and HERALD. The domains

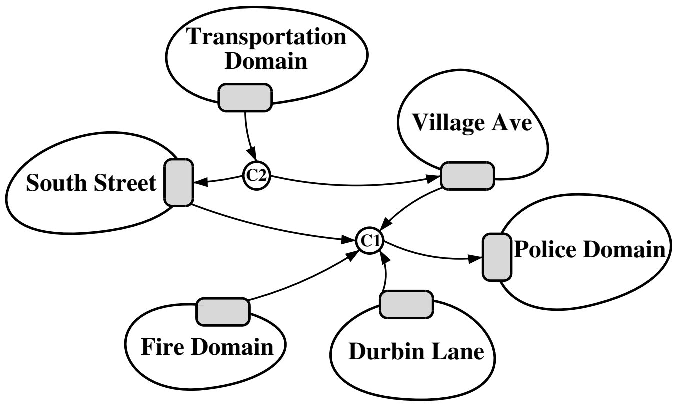


Fig. 6.8: An Active City

consist of a transportation information system, a police station and a fire station. Each of these domains was built using the HERALD event architecture. We have created two global contracts for the *Active City* application.

In the first scenario we consider a police station interacting with *Active Houses*. The police station monitors break-ins. In the second scenario we look at a transportation information system that can be used to monitor bus arrival times. We first describe the global contracts and then discuss some of the observer functions within the federations and domains that map the contract types for local observation. We will use the application shown in figure 6.8 as the basis for the discussion.

6.2.2 The City Contracts

The first city-wide global contract was created to deal with *burglar* events. In the *Active Street* application, the two initial houses were both able to produce a *break-in* event. The street had a global contract for these event types, which individual houses could join (see figure 6.3(b)). This contract was used to implement a *neighborhood watch* scheme. The global contract from the original street (and other streets like it) is the basis for the new global contract. This contract also illustrates how gateways can add information to contract types in a meaningful manner. The contract for *burglar* events is shown in figure 6.9(a). The attribute `city` is an extra attribute that the original *South Street* federation contract did not have. The function that maps the *South Street* break-in contract to the new *Active City* burglar contract generates the extra attribute. This is done by simply adding the name of the city into the XSLT translation function. For each street where the street contract type did not have a city element, the translation function needed to provide the extra attribute when joining

```
<complexType name="burglarEvent">
 <extension base="baseEvent">
  <sequence>
   <element name="street" type="xsd:string"/>
   <element name="city" type="xsd:string"/>
   <element name="room" type="xsd:string"/>
  </sequence>
 </extension>
</complexType>
```

(a) A Contract *break-in* Event

```
<complexType name="busArrivalEvent">
 <extension base="baseEvent">
  <sequence>
   <element name="busLine" type="xsd:string"/>
   <element name="busStop" type="xsd:string"/>
   <element name="eta" type="xsd:integer"/>
  </sequence>
 </extension>
</complexType>
```

(b) A Contract *busArrival* Event

Fig. 6.9: Global Contracts for the *Active City*

the *Active City* burglar contract.

The second contract in the city is used for monitoring bus arrival times. It enables individual entities within the city to monitor when buses are due to arrive at a particular bus station. The contract type is shown in figure 6.9(b). This example illustrates another instance where local domains/sub-federations did not need to join the contract before being able to observe contract types locally.

Federations/domains join the new burglar contract by providing a mapping function from the local type to the contract type. For the *Active Streets* this is a mapping function from the break-in contract type to the burglar contract type. For the *Fire* domain, this is a function from a local domain type to the burglar contract type. In this application the break-in event type of the *fire* domain is the same as the one produced by the original houses. The first contract (*C1* in figure 6.8) was joined by the three active streets and the fire domain. The police domain provided an observer function to the contract type. The second contract (*C2*) was only joined by the transportation domain. *South Street* and *Village Ave* provided observer functions for the contract type.

6.2.3 The Observer Functions

The police station provides a function to observe the burglar contract type locally. In doing so it is able to monitor all break-in events in the streets of the city that joined the contract.

Before *Active Houses* can make use of the federation level *busSchedule* contract, individual streets must provide an observer function for the contract type. Once this has been done, the contract type becomes locally observable. In other words, the contract

C2 in figure 6.8 becomes available for the houses of an *Active Street* once it has been made observable for the sub-federation. Making it observable within *South Street*, for example, is the same as introducing a global contract, of the observable type, in the street. Once this contract type has been created, local houses can provide an observer function for it and observe the new observer/contract type locally.

6.3 Connecting Heterogeneous Mobile Devices

In many instances it is more useful to receive information or event notifications without having to be in the vicinity of a static networked device. Receiving bus arrival times on an output device while at home is only useful if one is interested in catching a bus from a bus stop close to home. While shopping in the city, another means of information discovery is necessary. Ideally, one should be able to receive *busSchedule* events on a mobile device, such as a mobile phone or personal data assistant (PDA). Mobile devices of this kind have recently been used in tourist guide applications [DMCB98, CMD99, CDMS00]. These applications use wireless communication to enable them to create systems which provide mobile connectivity and context-awareness.

The event federation paradigm, particularly contracts and gateways is well suited to mobile device communication. Mobile devices connect to the network when at or near a network station or antenna. The device is connected as long as it remains within a certain radius of the antenna. Once outside the radius, the device is disconnected until it is once again within the radius of the same networked station or another one.

The characteristics of a station are similar to a gateway component in the federation paradigm, the major difference being that the gateway must support disconnected computing to be able to handle mobile entities. This means a gateway must store information for as long as a device remains disconnected. Once the device has reconnected, the information is routed to the device (possibly via another gateway). Thus, the gateway needs to store information, as well as re-route messages in case a device reconnects at a different gateway.

As in the case with networked devices, the gateway still manages event and subscription translations. Mobile devices register interest with a service such as the one provided by the *busSchedule* contract within the *Active City.* A mobile device is informed of bus arrival times as soon as this event becomes available. Because the service is asynchronous, the device need not be connected at all time. When disconnected, the notification is stored at the gateway until the device reconnects.

This idea can be used for other services, besides event notifications. Mobile devices may want to be informed about restaurants within a certain area of a city. This type of information is not suitable for transmission via the event paradigm. Rather than subscribing to an event service, the device sends a query to an information service. The information the service provides is likely to be in a format not suitable for each and every mobile device. The information may need to be translated at the gateway before being transmitted to the device. The gateway stores translation functions for particular devices and applies the mapping before the information is sent to a device. The same may happen for queries sent to an information service. The query from a particular device may need to be translated before being sent to the service. This information is also stored at the gateway.

6.4 Summary

In this chapter we have shown sample applications using the event federation architecture to link heterogeneous component event systems. The first application linked heterogeneous *Active Houses* to form an *Active Street*. The second application linked streets and other active entities to form an *Active City*. The final application showed how the event federation paradigm could be used to link mobile entities.

Active Houses are houses that have intelligent devices and appliances linked via a network. Devices and appliances can communicate with each other using event notification middleware. In the original *Active House* appliances are both event producers and consumers. An appliance can register interest with another appliance. Once an event occurs, an action can be triggered. For example, the stereo system can register interest in an *enter* event produced by the security system. When this event occurs an action such as `muteStereo` can be triggered.

The original house was built using a generic CEA-based event architecture that could not be dynamically extended with new event services. For this reason and to illustrate the event federation paradigm, we built a second house using the HERALD event architecture. These two houses were then linked via the federated architecture to create an *Active Street*. Two global contract types were generated for the street. Each house joined these contracts by providing a translation function from the local to the contract type. The two houses also created two domain-to-domain contracts. Observer functions for some of the contract types were provided by the HERALD house. This allowed events generated within the original *Active House* to be received within the HERALD house. A fire station was added to the street. It provided an observer function for one of the contract types. This allowed it to receive *fire* events from all houses that joined the global `fireEvent` contract.

The idea of the *Active Street* was extended to an *Active City*. Here federations of streets and other active entities (domains) were linked. Two global contracts were used to link different federations and domains within the new federation. This application illustrated how the event federation paradigm could be used to link existing federations.

The chapter concluded by showing how the concept of a federation linked by gateways can be used for communication between mobile entities. Gateways can manage disconnection and information translation, both for event instances and other information sources.

7. ANALYSIS AND FURTHER WORK

To evaluate the solutions presented in this thesis, it is helpful to look back at chapter 1 where we outlined the aim of this thesis. This was to develop and describe an architecture for connecting heterogeneous event-based middleware platforms. The research issues that the thesis addressed included developing an event federation paradigm and based on this paradigm designing an architecture for event federation. This architecture was then to be used to build applications linking heterogeneous event systems.

This chapter presents a synopsis of the work discussed in this thesis, evaluating its relevance and where appropriate highlighting directions for future research.

7.1 Analysis

An analysis must begin with a look at the original aim of the thesis. Based on this a critical evaluation of the work can be made. Above, we have outlined the initial research issues and aims of this thesis. An argument was made in chapter 1 that highlighted the need for heterogeneous event systems to be linked. This argument was based on the realization that information sharing is and will continue to be an important aspect of computer science. With the advent of wide-area computing, linking vastly distributed data sources and applications will become a reality. Asynchronous communication is the most suitable architecture for this. Event systems have emerged as the standard platform for this from of communication. The need to exchange awareness or location information has already become an issue of importance. We also argued that it is unlikely that a single standard will emerge for event based middleware communication. Based on these arguments we proposed a paradigm and an architecture based on this paradigm for linking heterogeneous event-based platforms.

In chapter 2 we described existing middleware platforms, highlighting the importance of metadata in such systems. This was used as a basis for the event federation paradigm, which required that component event systems are able to express their events through a formal schema.

In chapter 3 we described research in the area of federated database systems. Two areas were looked at in particular. The first was the five-level schema architecture. This architecture can lead to a tightly coupled system architecture because a central authority creates a global schema into which all members must map their data. The second approach, referred to as the wrapper/mediator approach, was shown to be less restrictive since participating systems are able to determine their own mappings depending on their needs. Furthermore, this approach tends to allow multiple federated schemas to coexist in a federation. Therefore, these systems are less tightly coupled and better suited for large scale integration of a large number of applications or data sources. The wrapper/mediator approach was therefore the more suitable method for integrating heterogeneous event systems.

Based on this research the thesis proposed a paradigm for integrating event systems. The paradigm was motivated by three factors.

1. Event systems produce information that must potentially be shared with other systems. Sharing this information requires resolving semantic heterogeneity.
2. Modern event systems describe their data through a schema. This means that information is typed. This is a key to being able to share information.
3. Resolving semantic heterogeneity has been widely studied in database research. These systems are generally tightly coupled. Tight coupling is possible when the number of information sources is small and the requirements are transparent from the beginning, but impossible with a very large number of component systems where the requirements are not clear from the outset and flexibility is important.

7.1.1 The Paradigm

The central components of the event federation paradigm are domains, contracts and gateways. Contracts bind domains, making them agree to translate a local type into a global contract type. Gateways manage the contracts and the translation functions. The notion of a contract type is closely related to the idea of a federation schema in the five level schema architecture [SL90]. Unlike in the five level schema architecture though, domains can form their own contracts with other domains. This is a more flexible approach than only permitting domains to map to a federated schema. Depending on the federation, a more tightly coupled system can be designed by not allowing domains to create new contracts. In such a federation, contracts are created by a global authority which monitors the creation of new types. The paradigm is able to incorporate both contract styles, making it more flexible than other types of federations.

Besides offering a flexible approach to system integration, the paradigm also provides a clear description of the functional requirements of a gateway component. This component is central to the architecture and links heterogeneous domains. The paradigm establishes the minimal requirements for a gateway. In the architecture, application developers are free to add extra functionality to the component when and if needed.

In trying to be flexible, the paradigm also makes minimal assumptions about the architecture of an event federation. It only prescribes five requirements for a federation (see section 4.1) and makes two assumptions about component event systems wishing to participate in a federation (see section 4.4). This not only allows a wide variety of event architectures to be integrated into a federation, but also gives the designer of a federation a great deal of flexibility.

7.1.2 The Architecture

The architecture of the gateway and the choice of data model for event representation were motivated by the paradigm, in the desire to be flexible and with the goal to provide easy interoperability. The global subscription language *FedReg* was designed with the potential to express all subscription styles encountered in existing event systems. Being able to express all subscription styles in the global subscription language did thus not limit the number of possible event systems that could be integrated. The same was true for the choice of XML and XML Schema for event instance and event type representation. The ability to express event types and their instances for a wide variety of event systems again meant that no unnecessary restrictions were placed on federation participants.

It was clear that a trade-off had to be made between flexibility and complexity. By making the data model and the subscription language very flexible and thereby allowing a wide variety of event-based systems into the federation, the complexity of the gateway increased. This was particularly true of the wrapper component of the gateway since it was responsible for resolving data model mismatches. By providing three interfaces for both the mediator and the wrapper, we limited the complexity somewhat. These interfaces allowed each component to treat the other as a black box.

Whether other data models are better suited for the event federation paradigm remains open. Our initial work investigated the use of *ODL* [CBB+99] for event type representation. This will be discussed in more detail in section 7.2.1.

Having opted for *gateway filtering* to resolve possible subscription language mismatches was again a design choice that tried to provide a reasonably scalable solution without limiting the number of component event systems that could participate in a federation. This is also a direction for further research and will be discussed in section 7.2.2.

7.1.3 The Experiments

The experiments described in chapter 6 use the gateway architecture to link heterogeneous event systems in the federated style described in this thesis. These experiments demonstrated that the event federation paradigm provides a feasible solution for linking heterogeneous event systems. The scale of the project limited the number of different event systems that could be integrated. Since the gateway architecture used HERMES for inter-domain communication, we essentially integrated three types of event platforms in the experiments.

One of the difficulties in the prototype applications was trying to simulate semantic heterogeneity. A possible solution would have been to let different users design different houses. This would have provided a more realistic scenario and would have been an opportunity to test extending event definitions with extra semantic information in the form of comments. Resolving semantic heterogeneity will always be a difficult problem and tools that will aid in the process are likely to emerge [Gru95, Kni93]. It is also likely that ontologies will be integrated into applications where semantic conflict resolution is necessary. This will be discussed further in section 7.2.1.

7.2 Further Work

In the above analysis we have outlined possible directions for further research. Although the thesis achieved its aims and provided a novel solution to event system integration, it uncovered many new and interesting questions. In this section we will discuss some of these ideas.

The first area we discuss is that of data description. Resolving semantic heterogeneity is a difficult problem. The more descriptive the data to be integrated, the more likely that semantic heterogeneity can be resolved. An interesting approach for achieving this is by the use of ontologies. In short, an ontology is a specification of a conceptualization. It represents more meaning than metadata itself. We will therefore first discuss some possible data models for event federations and then discuss how ontologies go a step further by providing information beyond metadata.

7.2.1 Data Model and Ontologies

A federation data model must be as expressive as possible. This guarantees that a wide variety of component data models can be represented in the federated data model. This was a criterion for our choice. Using XML Schema, we were guaranteed to be able to express most data models used by event based middleware platforms. Furthermore, it provided a standard syntax.

Another important consideration is whether a language is strongly or weakly typed. Strongly typed systems tend to be more robust, at the expense of extra computation either at run time or at compile time. Our initial research used ODMG's *ODL* for federation-wide data representation. Research in this area had already proposed extensions to the standard [Rad96, BFN94] to be able to express federated schemas in the style of the five level schema architecture. Besides a canonical data model, the standard also provides language bindings for C++, Java and Smalltalk.

Besides being able to fully map a wide variety of data models, it is also important that a global data model allows a component system to express detailed information about the facts it contains. In the five level schema architecture in chapter 3 the export schema describe the data the component system is willing to share with the federation. This schema uses the global data model for metadata representation. This is identical to component event systems publishing the local event types that they are willing to share with the federation. Here, an expressive data model may allow the use of added semantic information to more clearly identify the meaning of data definitions. In chapter 3 we described an architecture that used *semantic values* to express data in more detail. It was argued that this was a possible solution to aid in resolving semantic heterogeneity.

Ontologies present another possible solution for resolving or partially removing ambiguities in information. The term stems from philosophy where it referred to the subject of existence. Later it became popular in the field of *Artificial Intelligence* (AI), where it was used to identify a specification of a conceptualization.

Outside of AI, ontologies are mainly used to enable knowledge sharing. They have become popular in all fields where heterogeneous information is exchanged. One such area is *e-commerce* where data is exchanged between different entities such as suppliers, retailers and consumers. Retailers require data to flow from wholesalers and wholesalers require data to flow from producers. Data exchange of this kind used to consist of tab-delimited dumps or product-specific tables. Using specific XML formats for each exchange task has improved the situation, but it is very cumbersome and only provides a solution for a specific instance. Approaches like *ebXML* [KW02] try to tackle this problem by aiming to develop an open XML-based infrastructure.

Ontologies provide a solution for representing the different terminologies used in different types of businesses. They enable information and knowledge sharing between different applications through a shared set of terms describing the application domain with a common understanding. An ontology not only provides a flat set of terms (much like a schema) but also describes relationships between these terms. This helps an application to at least partially understand the domain. They do this by establishing a joint terminology between members of a community of interest. These members can be humans, agents or applications. A common ontology defines the vocabulary with which queries, assertions or information is exchanged among programs. Ontological commitments are agreements to use the shared vocabulary in a coherent and consistent manner. A commitment to a common ontology is a guarantee of consistency, but not completeness, with respect to queries, assertions and information, using the vocabulary defined in the ontology.

To represent a conceptualization a representation language is required. For applications it is important to have a language with a standardized syntax. Since XML has emerged as a standard language for data interchange, exchanging ontologies using an XML syntax has become popular. Examples of such languages include Ontology Exchange Language (*XOL*), Ontology Markup Language (*OML*) [OHS01] and Resource Description Framework Schema Language (*RDFS*) [LS99, Con02a]. All of the languages use XML syntax, but with slightly different tag names.

Developing an ontology for application domains where event notification is the primary means of information exchange is thus clearly a direction for further research. Some projects have begun to tackle this area of research [CBB01], but it is still in its infancy.

7.2.2 Subscription Language

The subscription language used by a federation should be expressive enough to represent a wide range of subscription styles. If the global subscription language is less expressive than a component system's subscription language, a filter must be placed at the point where the subscription translation occurs from the more to the less expressive language. In the architecture described in this thesis, we made the assumption that the federation's subscription language is always more expressive than the component system's subscription language. This meant the architecture had to provide *gateway filtering* at the gateway of the domain for which the subscription was intended. If the federation's subscription language was less expressive than any of the component event systems' languages, the filter would have had to be placed at the gateway of the domain where the subscription originated. This would mean that event instances not necessarily of interest to a domain would be transmitted across the domain boundary, thus unnecessarily increasing network traffic.

The federated event architecture described in this thesis opted to provide *gateway filtering*, which meant that the gateway component increased in complexity. A complete survey of subscription styles may make it possible to determine an ideal subscription language for a set of component event systems. With such a set of component event systems it may be possible to provide true source-side filtering. Some surveys exist for event notification systems [RK98], but they focus less on the subscription style of these systems.

Although *FedReg* was an expressive language and provided support for complex filtering, including equality, inequality, logical operators and regular expression matching on strings, it was not as versatile and clean as it could have been. The filter types in templates were not properly typed. Rather, they were of type *string* (see figure 5.2(b)) and thus needed to be parsed for every evaluation. This added to the complexity of the filtering module in the gateway. A cleaner, but more complex, solution would have been to properly type all filter expressions.

7.2.3 Security

Many security issues arose in the course of the project. By using identifiers handed out by a federation authority to uniquely identify each domain (and their messages) within the federation, the architecture provided only a limited form of security. In many instances a more fine-grained level of security would have been desirable. Role-based access control systems [Hay96] are well suited for this purpose. By using such a system, entities within a federation can be given individual roles which determine the level of privileges they have within the federation.

In the federated architecture certain entities only wanted to observe contract types, rather than join them. The police domain in section 6.2, for example, monitored break-ins by providing an observer function for the federated contract. Without a proper security mechanism it is not possible to keep other entities from providing an observer function for that contract when they have a proper domain identifier.

In other instances it is useful to restrict domains from being able to join a contract or form a contract with certain published event types from other domains. An *Active House* may only want to make the *doorbellRing* event type available to a particular neighbor, rather than the entire street. Again, a role-based access control system integrated into the gateway component could provide such security.

7.2.4 Composite Events

The event federation paradigm has only dealt with simple events. It has not discussed composite event detection [GJS92, LF98] at a federated level. A composite event comprises primitive and composite events combined by event operators. A composite event service carries out live detection of composite events expressed in a string-based syntax where a substring can represent a primitive or composite event from a single or combination of event sources. A simple algebra for composite event expressions is shown in figure 7.1. The operators can be combined to form complex expressions.

Operators		Expressions	Pattern matched
;	FOLLOWED BY	A ; B	an A followed by a B
|	OR	A | B	any A or B
&	AND	A & B	an A and a B in either order
−	WITHOUT	A − B	the stream of As until a B occurs
First	THE FIRST OF	First(A)	the first of a possible stream of As

Fig. 7.1: A Composite Event Algebra

Once detected, a composite event is given the timestamp of the latest constituent event. As time is relativistic by nature in distributed systems and the order of events is generally important, special attention must be paid to determining when an event has occurred. [LCB99] introduces the concept that an event happened in some source-dependent, UTC-time interval, and incorporates the property of time-consistent order on events. The use of intervals allows the duration of a composite event to be recorded in the timestamp.

Many event notification architectures support composite event detection. This means that a client can register with a composite event in the local subscription style within a domain. Currently this is only possible for events that occur within the local domain of that component event system. Translation of composite event subscriptions is currently not supported by the federated paradigm and is an interesting direction for further research. Many scenarios are possible. Some of these are listed below.

1. A local domain subscribes to a composite event where the individual events come from a single foreign domain which supports composite event subscription.
2. A local domain subscribes to a composite event where the individual events come from a single foreign domain, which does not support composite event subscription.
3. A local domain subscribes to a composite event where the individual events come from multiple domains, which may or may not support composite event detection.
4. A local domain subscribes to a composite event where the individual events come from the local as well as a foreign domain, where the foreign domains may or may not support composite event detection.

These four possibilities highlight the difficulty involved in supporting composite event detection at a federated level. The first and second scenarios are simpler to implement than the third and fourth. In both instances the gateway could provide the composite event detection engine. The approach would be similar to *gateway filtering*. In the first scenario it may also be possible to translate the local subscription to a global subscription and then into a subscription for the foreign domain. This would be possible if both subscription languages support the same operators and the global subscription language is extended to include composite event expressions. In that instance the foreign component event system could manage the composite event detection. A combination of these two approaches is possible for the first scenario.

The third scenario is complex and it appears that one solution is to have the local domain's gateway provide composite event detection for events coming from foreign domains making up a composite event. This would require the gateway to decompose complex expressions into simple event subscriptions. These are then sent to the foreign domains. Individual events are then received by the local gateway and interpreted accordingly.

The final scenario is the most complex to implement and cannot be implemented solely at the gateway level. Since an expression may contain events from within the local domain, the component event system may need to be extended with a module that can decompose expressions into individual subscriptions and then pass those to the gateway. All composite events are then detected within the local domain when event instances are passed back to the local domain.

Clearly, there are trade-offs in the design of an implementation of federated composite event detection. One would like to provide filtering as close to the source as possible. With each of scenarios described above, the filtering moved further away from the source.

7.2.5 Further Thoughts

Another area for further research includes identifying all types of middleware systems that can possibly be incorporated in the event federation paradigm. Although chapter 4 mentions two characteristics an event or middleware platform must possess in order to be able to join a federation, a more detailed study would be useful. Obviously the more detailed the list, the less complex the gateway module becomes. Complexity is added to a gateway component when trying to include as many systems as possible.

This leads directly to another important question which resulted from the research, namely how much functionality should be incorporated into the gateway and how much should be provided by the component event systems. One could argue that a gateway may be responsible for providing filtering of subscriptions to a certain degree. If a minimal requirement could be defined for domains wishing to participate in the federation, less functionality would be required by the gateway. Again, there is a trade-off between trying to incorporate as many component systems as possible and the complexity of the gateway component.

7.3 Summary

An analysis of the work described in this thesis showed that the overall aims were achieved. The goal was to develop a paradigm for federating event-based middleware. This was done in chapter 4. The paradigm was motivated by research in the areas of event-based middleware and database federation. Based on this paradigm, an architecture was developed. The architecture was described in chapter 5. Chapter 6 described an experimental framework based on this architecture.

As in any project of this size, new questions surfaced during the research and development phases, which gave directions for further research. These directions were discussed in this chapter. One area of interest was the data model used by the federation. A data model can be very helpful in resolving semantic ambiguity. Another possibility besides a very descriptive data model is the use of an ontology. An ontology is a specification of a conceptualization that can remove ambiguity by using an agreed vocabulary for data exchange. Ontologies have been widely studied in *e-commerce* and it appears that their use may be of interest for federating event-based middleware.

The subscription language used by the federation was also a possible area for further research. We showed how the subscription language determined where filtering had to occur within the federation. Filtering as close to the source as possible meant that a filter needed to be placed at the point where the translation from a more to a less descriptive language occurred. The choice of a subscription language is thus influenced by the requirements to provide source-side filtering and the complexity of the federation components, particularly the gateway.

Domain security is an important issue for an event federation. The federated paradigm discussed the notion of a federation authority. This authority distributes domain identifiers which uniquely identify each domain within a federation. The identifiers were used to tag every message sent between domains. This provided a weak form of security. More complex scenarios are desirable, which can possibly be provided by role-based access control systems. This would allow for a more fine-grained level of security, giving different participants in the federation different rights. In such a scenario, domains could be restricted individually from subscribing to particular event types or joining certain contracts. This would also be a useful mechanism to determine which domains can provide observer functions and which cannot.

The chapter concluded by considering clearer guidelines to determine which types of systems can be incorporated into the federated paradigm. By restricting certain types of systems, the development of gateways would become simpler. As an example, we showed how the subscription language could be tailored to a specific set of platforms.

8. CONCLUSION

This dissertation began by describing a scenario in which mobile devices are linked in an *Active City* and able to communicate with each other and with other event-based information sources. This was a futuristic scenario since communication of heterogeneous event platforms has not yet become a reality. Motivated by three factors, the thesis proposed an event federation paradigm. This paradigm is motivated by research in event notification and messaging systems as well as federated systems. Based on these ideas and the realization that asynchronous communication using an event notification platform provides the most appropriate means of connecting vastly distributed systems, the dissertation proposed a paradigm for linking event-based middleware platforms. The dissertation describes an architecture based on the paradigm. The architecture was tested in an experimental framework. Based on these experiments, the dissertation drew conclusions about the event federation paradigm and its implementation.

It was argued that it is unrealistic to assume that a unified standard for event communication will evolve. At the same time, the need to share information is likely to increase in the future. Sharing information has been and will continue to be an important aspect of everyday life. This has been the case for a number of applications and information sources, particularly databases. Progress has been made in these areas, but no research has ever tackled linking heterogeneous event systems.

Event-based middleware has become the *de facto* standard for connecting heterogeneous applications and for providing awareness support. Events and structured messages are pieces of information used for communication between entities of such systems. Applications built on top of different event-based middleware platforms cannot communicate with each other because of system and structural semantic heterogeneity. In many instances communication is nonetheless desirable and to achieve this an architecture for resolving heterogeneity is required.

The paradigm described in this thesis is based on three principle components. They are domains, contracts and gateways. Gateways link domains via contracts. A domain is a component event system with its own administrative authority. A contract is an agreement to translate local event instances from one domain into a type defined by the contract. The translation function is part of this contract. Gateways mange these contracts and apply the translation functions to event instances, subscriptions and event definitions. Based on these ideas we built gateways that connect different event architectures via contracts.

The architecture was tested in experimental applications. The first application linked heterogeneous *Active Houses*, houses with intelligent networked devices, to form an *Active Street*. The application was extended and *Active Streets*, along with other event-based domains, were linked to form an *Active City*. Besides demonstrating the validity of the event paradigm , these applications showed how domains publish local event types, event instances are mapped to contract types, cross-domain subscriptions are translated and contract types are observed locally via observer functions.

An analysis of the work highlighted directions for future research. These areas included the possible use of ontologies to describe event services, an evaluation of possible global subscription languages and role-based access control at the gateway level.

APPENDIX

A. SCHEMA FOR INTER-DOMAIN MESSAGES

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns='http://www.w3.org/2002/XMLSchema'
        xmlns:h="http://www.cl.cam.ac.uk/opera/eventfed"
        targetNamespace='http://www.cl.cam.ac.uk/opera/eventfed'>

  <element name="hermes" minOccurs="1" maxOccurs="1">
    <complexType>
      <choice>
        <group ref="h:broadcast"/>
        <group ref="h:msgType"/>
      </choice>
      <attributeGroup ref="h:messageAddress"/>
      <attribute name="domain" type="positiveInteger"/>
    </complexType>
  </element>

  <group name="broadcast">
    <sequence>
      <element name="broadcast">
        <complexType>
          <sequence>
            <group ref="h:msgType"/>
          </sequence>
          <attributeGroup ref="h:messageAddress"/>
        </complexType>
      </element>
    </sequence>
  </group>

  <group name="msgType">
    <choice>
      <element name="control">
        <complexType>
          <choice>
            <element name="request">
              <complexType/>
            </element>
            <group ref="h:acceptreject"/>
          </choice>
          <attribute name="type">
            <simpleType>
              <restriction base="string">
                <enumeration value="broker"/>
                <enumeration value="source"/>
                <enumeration value="sink"/>
                <enumeration value="gateway"/>
        </restriction>
      </simpleType>
    </attribute>
  </complexType>
</element>
<element name="routing">
  <complexType>
    <sequence>
      <element name="entry" minOccurs="1" maxOccurs="unbounded">
        <complexType>
          <attributeGroup ref="h:messageAddress"/>
          <attribute name="cost" type="positiveInteger"/>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
<element name="type">
  <complexType>
    <choice>
      <element name="addtype">
        <complexType>
          <sequence>
            <any namespace="http://www.w3.org/2000/10/XMLSchema" minOccurs="1"
              maxOccurs="unbounded" processContents="lax"/>
          </sequence>
          <attribute name="typename" type="string"/>
          <attribute name="extends" type="string"/>
        </complexType>
      </element>
      <group ref="h:acceptreject"/>
    </choice>
  </complexType>
</element>
<element name ="advertisement">
  <complexType>
    <choice>
      <element name="advertise">
        <complexType>
          <attribute name="typename" type="string"/>
        </complexType>
      </element>
      <group ref="h:acceptreject"/>
    </choice>
  </complexType>
</element>
<element name="subscription">
  <complexType>
    <choice>
      <element name="subscribe">
        <complexType>
          <sequence>
            <element name="fedreg" type="string"/>
          </sequence>
          <attribute name="typename" type="string"/>
        </complexType>
      </element>
      <group ref="h:acceptreject"/>
    </choice>
  </complexType>
</element>
<element name="notification">
  <complexType>
    <choice>
      <element name="notify">
        <complexType>
          <sequence>
            <any namespace="http://www.cl.cam.ac.uk/opera/eventfed/types"
             minOccurs="1" maxOccurs="unbounded" processContents="strict"/>
          </sequence>
          <attribute name="typename" type="string"/>
          <attribute name="subscription" type="string" use="optional"/>
        </complexType>
      </element>
      <group ref="h:acceptreject"/>
    </choice>
  </complexType>
</element>
<element name="contract">
  <complexType>
    <choice>
      <element name="contracting">
        <complexType>
          <sequence>
             <any namespace="http://www.w3c.org/1999/XSL/Transform"
              minOccurs="1" maxOccurs="unbounded" processContents="strict"/>
          </sequence>
          <attribute name="typename" type="string"/>
        </complexType>
      </element>
      <group ref="h:acceptreject"/>
    </choice>
  </complexType>
</element>
<element name="observer">
  <complexType>
    <choice>
      <element name="observe">
        <complexType>
          <sequence>
            <any namespace="http://www.cl.cam.ac.uk/opera/eventfed/types"
             minOccurs="1" maxOccurs="unbounded" processContents="strict"/>
            <any namespace="http://www.w3c.org/1999/XSL/Transform"
             minOccurs="1" maxOccurs="unbounded" processContents="strict"/>
          </sequence>
          <attribute name="contracttypename" type="string"/>
        </complexType>
      </element>
      <group ref="h:acceptreject"/>
    </choice>
  </complexType>
</element>
<element name="discovery">
  <complexType>
```

```
          <choice>
            <element name="discover"/>
            <group ref="h:acceptreject"/>
          </choice>
        </complexType>
      </element>
      <element name="discoveryreply">
        <complexType>
          <choice>
            <element name="discoverreply">
              <complexType>
                <sequence>
                  <any namespace="http://www.cl.cam.ac.uk/opera/eventfed/types"
                   minOccurs="1" maxOccurs="unbounded" processContents="strict"/>
                </sequence>
              </complexType>
            </element>
            <group ref="h:acceptreject"/>
          </choice>
        </complexType>
      </element>
    </choice>
  </group>

  <group name="acceptreject">
    <choice>
      <element name="accept">
        <complexType/>
      </element>
      <element name="reject">
        <complexType/>
      </element>
    </choice>
  </group>

  <attributeGroup name="messageAddress">
    <attribute name="name" type="string"/>
    <attribute name="port" type="positiveInteger"/>
  </attributeGroup>

</schema>
```

BIBLIOGRAPHY

[5T02] 5T. Telematics Technologies for Transport and Traffic in Turin. http://www.5t-torino.it/home_en.html, 2002.

[AAAF01] F. Ahmad, S. Ahmad, Z. Ahmad, and U. Farooq. Experimenting with Jini on Linux. *Embedded Linux Journal*, 1:45–48, 50–52, January/ February 2001.

[ASS+99] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-based Subscription System. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–62, New York, May 1999. Association for Computing Machinery.

[Bat96] J. Bates. *Presentation Support for Distributed Multimedia Applications.* PhD thesis, University of Cambridge, 1996.

[BB99] C. Bornhövd and A. P. Buchmann. A Prototype for Metadata-based Integration of Internet Sources. In M. Jarke and A. Oberweis, editors, *Proceedings 11th Int. Conf. Advanced Information Systems Engineering, CAiSE*, number 1626 in Lecture Notes in Computer Science, LNCS, pages 439–445. Springer-Verlag, June 1999.

[BBHM96] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[BBMS98] J. Bacon, J. Bates, K. Moody, and M. Spiteri. Using Events for the Scalable Federation of Heterogeneous Components. In *8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

[BFN94] R. Busse, P. Fankhauser, and E. J. Neuhold. Federated Schemata in

ODMG. In *First International East/West Database Workshop*, pages 356–379. Springer-Verlag, Berlin Germany, 1994.

[BHM$^+$00] J. Bacon, A. Hombrecher, C. Ma, K. Moody, and W. Yao. Event Federation and Storage using ODMG. In *Proceedings of the 9th Internationl Workshop on Persistent Object Systems: Design, Implementation and Use (POS9)*, pages 265–281, Lillehammer, Norway, September 2000.

[BKS$^+$99] G. Banavar, M. Kaplan, K. Shaw, R. Strom, D. Sturman, and W. Tao. Information Flow Based Event Distribution Middleware. In *Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999.

[Bla87] M. Blakey. Basis of a Partially Informed Distributed Database. In *Proceedings of the 13th International Conference on Very Lage Databases (VLDB)*, Brighton, UK, September 1987.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4), December 1986.

[BMB$^+$00] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic Support for Distributed Applications. *IEEE Computer*, pages 68–77, March 2000.

[Buc99] A. P. Buchmann. Architecture of Active Database Systems. In N. W. Paton, editor, *Active Rules in Database Systems*, Monographs in Computer Science, chapter 2, pages 29–48. Springer, 1999.

[Cat91] R. G. G. Cattell. *Object Data Management.* Addison-Wesley, Reading, MA, 1991.

[CBB$^+$99] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann Publishers, San Diego (CA), USA, 1999.

[CBB01] M. Cilia, C. Bornhövd, and A. P. Buchmann. Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments. *Lecture Notes in Computer Science*, 2172:195–208, 2001.

[CDMS00] K. Cheverst, N. Davies, K. Mitchell, and P. Smith. Providing Tailored (Context-Aware) Information to City Visitors. *Lecture Notes in Computer Science*, 1892:73–79, 2000.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, number 20 in VLDB, Santiago, Chile, 1994. Morgan Kaufmann Publishers, Inc.

[CMD99] K. Cheverst, K. Mitchell, and N. Davies. Design of an Object Model for a Context Sensitive Tourist GUIDE. *Computers and Graphics*, 23(6):883–891, December 1999.

[Col97] R. M. Colomb. Impact of Semantic Heterogeneity on Federating Databases. *The Computer Journal*, 40(5), 1997.

[Con99] World Wide Web Consortium. Namespaces in XML, W3C Recommendation, January 1999. http://www.w3.org/TR/1999/REC-xml-names-19990114.

[Con00] World Wide Web Consortium. Extensible Markup Language (XML) 1.0, (Second Edition), W3C Recommendation, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[Con01a] World Wide Web Consortium. XML Schema Part 0: Primer, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502.

[Con01b] World Wide Web Consortium. XML Schema Part 1: Structures, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502.

[Con01c] World Wide Web Consortium. XML Schema Part 2: Datatypes, May 2001. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

[Con02a] World Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/2002/WD-rdf-schema-20020430, April 2002.

[Con02b] World Wide Web Consortium. XML Path Language (XPath) Version 1.0, April 2002. http://www.w3.org/TR/1999/WD-xpath20-20020430.

[Con02c] World Wide Web Consortium. XSL Transformations (XSLT) Version 2, April 2002. http://www.w3.org/TR/2002/WD-xslt20-20020430.

[Cor99] Microsoft Corporation. Message Queue Server Reviewer's Guide. Technical report, Microsoft Corporation, 1999. http://www.microsoft.com/ntserver/techresources/appserv/MSMQ/msmqrevguide.asp.

[CRW00] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–228, New York, USA, July 2000. ACM Press.

[DM01] D. J. Dailey and S. Maclean. Smart Trek: Busview. http://www.its.washington.edu/projects/busview_overview.html, 2001.

[DMCB98] N. Davies, K. Mitchell, K. Cheverst, and G. Blair. Developing a Context Sensitive Tourist Guide. In *Proceedings of the First Workshop on Human Computer Interaction with Mobile Devices*, Glasgow, Scotland, 1998.

[DWMC00] D. J. Dailey, Z. R. Wall, S. D. Maclean, and F. W. Cathey. An Algorithm and Implementation to Predict the Arrival of Transit Vehicles, October 2000. ITSC2000 Conference.

[Fal00] E. Fallon. Dublin Bus Tracking Service: Design and implementation of a device independent passenger information system. Master's thesis, Trinity College, Dublin, November 2000.

[FCHH99] M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC(TM) API Tutorial and Reference*. Addison-Wesley Publishing Company, 2 edition, June 1999.

[FLPS00] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Efficient Matching for Content-based Publish/Subscribe Systems. Technical report, INRIA, 2000. http://wwwcaravel.inria.fr/pereira/matching.ps.

[FMK+99] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, T. Phelps, and B. Segall. Augmenting the Workaday World with Elvin. In Susanne Bødker, Morten Kyng, and Kjield Schmidt, editors, *Proceedings of the Sixth European Conference on Computer Support Cooperative Work (ECSCW-99)*, pages 431–450, Dordrecht, NL, September 1999. Kluwer Academic Publishers.

[Fou92] Open Software Foundation. *Introduction to OSF/DCE*. Prentice Hall, New Jersey, 1992.

[GHM96] J. C. Grundy, J. G. Hosking, and W. B. Mugridge. Towards a Unified Event-based Software Architecture. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international* *workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 121–125. ACM Press, 1996.

[GJS92] N. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases. In *Proceedings of the 18th Conference on Very Large Databases.* Morgan Kaufmann Publishers, Inc., August 1992.

[GKP99] R. E. Gruber, B. Krishnamurthy, and E. Panagos. The Architecture of the READY Event Notification Service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, Austin, Texas, USA, May 1999.

[GKP00] R. E. Gruber, B. Krishnamurthy, and E. Panagos. READY: A High Performance Event Notification Service. In *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[Gru95] T. R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies*, 43(5/6):907–928, 1995.

[GSSC96] M. Garcia-Solaco, F. Saltor, and M. Castellanos. *Semantic Heterogeneoty in Multidatabase Systems.* Prentice Hall, 1996.

[Hay96] Richard Hayton. *OASIS: An Open Architecture for Secure Interworking Services.* PhD thesis, Univeristy of Cambridge Computer Laboratory, June 1996. Technical Report No. 399.

[HBS+02] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Messaging Service, Version 1.1, April 2002.

[HH94] A. Harter and A. Hopper. A Distributed Location System for the Active Office. *IEEE Network*, 8(1), January/February 1994.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 169–180, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[HM85] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[HM93] J. Hammer and D. McLeod. An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems. *Journal for Intelligent and Cooperative Information Systems*, 2(1):51–83, 1993.

[HM98] A. Hombrecher and A. McNeil. The Active House Project. http://www.cl.cam.ac.uk/Research/SRG/opera/projects/active-house, 1998.

[Hol92] I. M. Holland. Specifying Reusable Components Using Contracts. In O. L. Madsen, editor, *Proceedings of the European Conference on Object-oriented Programming ECOOP*, LNCS 615, pages 287–308, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[Hul97] R. Hull. Managing Semantic Heterogeneity in Databases: A Theoretical Prospective. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 51–61, New York, NY 10036, USA, 1997. ACM Press.

[IBM99a] IBM. MQSeries Integrator: A Technical Overview. Technical report, IBM, 1999. http://www-4.ibm.com/software/ts/mqseries/library/whitepapers/eai-tech/.

[IBM99b] IBM. MQSeries Overview. Technical report, IBM, 1999. http://www-3.ibm.com/software/ts/mqseries/library/whitepapers/mqover.

[Inc99] Tibco Software Inc. TIB - Active Enterprise. Technical report, Tibco Software Inc., 1999.

[Ins75] American National Standards Institute. Interim Report of the ANSI/X3/SPARC Study Group on Data Base Management Systems. *ACM SIGFIDET*, 7:3–139, 1975.

[Jav97] JavaSoft. JavaBeans: Version 1.0.1, July 1997.

[KL98] A. Koschel and P. C. Lockemann. Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. *Journal of Data and Knowledge Engineering (DKE)*, 25, 1998. Special Issue for the 25th Vol. of DKE.

[Kni93] K. Knight. Building a large ontology for machine translation. In *ARPA Workshop on Human Language Technology*, March 1993.

[KW02] A. Kotok and D. R. R. Webber. *ebXML: The New Global Standard for Doing Business Qver the Internet.* New Riders Publishing, Carmel, IN, USA, 2002.

[LA86] W. Litwin and A. Abdellatif. Multidatabase Interoperability. *Computer*, 19(12):10–18, December 1986.

[LCB99] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-Dependent Distributed Systems. In *CoopIS'99: Fourth IFCIS Conference Cooperative Information Systems*, Piscataway, N.J., 1999. IEEE Press.

[LF98] D. C. Luckham and B. Frasca. Complex Event Processing in Distributed Systems. Technical Report CSL-TR-98-754, Stanford University, March 1998.

[LP98] S. L. Lo and S. Pope. The Implementation of a High Performance ORB over Multiple Network Transports. In Nigel Davies, Kerry Raymond, and Jochen Seitz, editors, *Middleware 98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware, pages 157–172. Springer-Verlag London Limited, www.springer.co.uk, September 1998.

[LS99] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. http:// www.w3.org/ TR/ REC-rdf-syntax, February 1999.

[LSPR93] E. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity Identification in Database Integration. *Information Sciences*, 89(1):1–38, 1993.

[MB98] C. Ma and J. Bacon. COBEA: A CORBA-based Event Architecture. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pages 117–132, Berkeley, April 1998. USENIX Association.

[MD89] D. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 215–224, 1515 Broadway, New York, NY 10036, USA, 1989. Association for Computing Machinery.

[MFO90] D. McKay, T. Finin, and A. O'Hare. The Intelligent Database Interface: Integrating AI and Database Systems. In William Dietterich, Tom; Swartout, editor, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 677–684. MIT Press, 1990.

[Mic95] Sun Microsystems. ONC Developer's Guide, November 1995.

[Mic01] Sun Microsystems. Jini Architecture Specification: Version 1.2, December 2001.

[NEC98] NEC. Notification Service (Joint Revised Submission). OMG TC Document telecom/98-01-01, January 1998.

[Net] Microsoft Developers Network. Open Database Connectivity (ODBC). http://www.microsoft.com/Data/odbc/faq_odbc.htm.

[Nor96] K. North. Understanding ODBC 3.0 standards and OLE DB. *DBMS*, 9(4):S15–S18, April 1996.

[Obj98] Object Management Group, OMG Headquarters, Framingham, MA, U.S.A. *The Common Object Request Broker: Architecture and Specification, revision 2.2*, omg 98-02-33 edition, February 1998.

[Obj00] Object Management Group - OMG. *Notification Service Specification*, June 2000.

[OHS01] A. L. Opdahl and B. Henderson-Sellers. Grounding the OML metamodel in ontology. *The Journal of Systems and Software*, 57(2):119–143, June 2001.

[OMG00] OMG. The Common Object Request Broker: Architecture and Specification, Revison 2.4. Specification, OMG, 2000.

[OMG01] OMG. Event Service Specification. Technical report, Object Management Group, March 2001.

[PB02] P. R. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, June 2002.

[Pie02] P. R. Pietzuch. Event-Based Middleware: A New Paradigm for Wide-Area Distributed Systems? 6th CaberNet Radicals Workshop, February 2002.

[Pri99] W. Prinz. NESSIE: An Awareness Environment for Cooperative Settings. In S. Bødker, M. Kyng, and K. Schmidt, editors, *Proceedings of the Sixth European Conference on Computer Support Cooperative Work (ECSCW-99)*, pages 391–410, Dordrecht, NL, September 1999. Kluwer Academic Publishers.

[Rad96] E. Radeke. Extending ODMG for Federated Database Systems. In Roland R. Wagner and Helmut Thoma, editors, *Seventh International Workshop on Database and Expert Systems Applications, DEXA '96, Proceedings*, pages 304–312, Zurich, Switzerland, September 1996. IEEE Computer Society Press, Los Alamitos, California.

[RCF98] M. Rosen, D. Curtis, and D. Foody. *Integrating COM and CORBA Applications.* John Wiley & Sons, Inc., 605 Third Avenue, New York, N.Y. 10158-0012, September 1998.

[RDR98] D. Ramduny, A. Dix, and T. Rodden. Exploring the Design Space for Notification Servers. In *Proceedings of ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, Primitives for Building Flexibile Groupware Systems, pages 227–235, 1998.

[Red97] F. E. Redmond. *DCOM: Microsoft Distributed Component Object Model.* IDG Books Worldwide, September 1997.

[RK98] A. Rifkin and R. Khare. The Evolution of Internet-scale Event Notification: Past, Present and Future, August 1998. http://www.cs.caltech.edu/adam/isen/wacc.

[Rou02] P. Rousselle. Implementing the JMS publish/subscribe API. *Dr. Dobb's Journal of Software Tools*, 27(4):28, 30–32, April 2002.

[RS93] R. Rabenseifner and A. Schuch. Comparison of DCE RPC, DFN-RPC, ONC and PVM. In Alexander Schill, editor, *DCE — the OSF distributed computing environment: client/server model and beyond: International DCE Workshop, Karlsruhe, Germany, October 7–8, 1993: proceedings*, Lecture Notes in Computer Science, pages 39–46. Springer-Verlag, 1993.

[RW97] D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale event Observation and Notification. In M. Jazayeri and H. Schauer, editors, *Proceedings 6th European Software Eng. Conf. (ESEC 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 344–360. Springer-Verlag, Berlin, 1997.

[SAB$^+$00] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proceedings of AUUG2K*, Canberra, Australia, June 2000.

[SBC$^+$98] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow based Approach to Message Brokering, 1998.

[SCGS91] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of Data Models as Canonical Models for Federated Databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(4):44–48, December 1991.

[Sei99] O. Seidel. *Metadata Support for Connecting Application Components Asynchronously.* PhD thesis, Computer Laboratory, University of Cambridge, 1999.

[SL90] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[Spi00] M. D. Spiteri. *An Architecture for the Notification, Storage and Retrieval of Events.* PhD thesis, Computer Laboratory, University of Cambridge, January 2000.

[SSR94] E. Sciore, M. Siegel, and A. Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database Systems*, 19(2):254–290, June 1994.

[Sun] Sun Microsystems. Java RMI. http://java.sun.com/products/jdk/rmi/.

[TK78] D. Tsichritzis and T. Klug. The ANSI/X3/SPARC Framework, 1978.

[WAC$^+$93] D. Woelk, P. Attie, P. Cannata, G. Meredith, A. Sheth, M. Singh, and C. Tomlinson. Task Scheduling using Intertask Dependencies in Carnot. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, May 26–28, 1993*, volume 22(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 491–494, New York, NY 10036, USA, 1993. ACM Press.

[WG97] G. Wiederhold and M. R. Genesereth. The Conceptual Basis for Mediation Services. *IEEE Expert*, 12(5):38–47, 1997.

[WHFG92] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society*, 1992.

[WM89] Y. R. Wang and S. E. Madnick. The Inter-Database Instance Identification Problem in Integrating Autonomous Systems. In *Proceedings IEEE International Conference on Data Engineering*, page 46, Los Angeles, CA, February 1989.