# An Architecture for the Notification, Storage and Retrieval of Events

Mark David Spiteri

Darwin College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

January 2000

# Abstract

*Event-driven* and messaging infrastructures are emerging as the most flexible and feasible solution to enable rapid and dynamic integration of legacy and monolithic software applications into distributed systems. They also support deployment and enhancement of traditionally difficult-to-build active systems such as large-scale collaborative environments and mobility aware architectures. However, complex systems issues like mobility, scalability, federation and persistence indicate a requirement for more advanced services within these infrastructures. The event notification paradigm is also applicable in emerging research areas such as modelling of business information flow within organisations, as well as workplace-empowering through enhanced awareness of work practices relating to communication and interaction between individuals. In these areas, further developments require complex interpretation and correlation of event information, highlighting the need for an event storage and retrieval service that provides the required groundwork.

It is the thesis of this dissertation that the lack of a generic model for event representation and notification has restricted evolution within event-driven applications. Furthermore, in order to empower existing applications and enable novel solutions, a crucial, and so-far-missing, service within event-driven systems is *capture, persistent storage, and meaningful retrieval* of the messaging information driving these systems.

In order to address these issues, this dissertation defines a *generic event model* and presents a powerful *event notification infrastructure* that, amongst other structural contributions, embeds event storage functionality. An *event repository* architecture will then be presented that can capture and store events, as well as *inject* them back into distributed application components to simulate replay of sequences of activity. The general-purpose architecture presented is designed on the thesis that events are temporal indexing points for computing activities. Changes in the state of a distributed system can be captured as events, and replayed or reviewed at a later stage, supporting fault-tolerance, systems management, disconnected operation and mobility. The architecture delivers powerful querying of event histories, enabling extraction of simple and composite event patterns. This addresses the business requirement in several industries (such as finance, travel, news, retail and manufacturing) to locate temporal patterns of activity, as well as support applications like memory prosthesis tools and capture of collaboration. The repository offers a selective store-and-forward functionality that enables messaging environments to scale and provide enhanced brokering and federation services.

In addition to enabling novel applications, the general-purpose infrastructure presented provides a more flexible approach to event notification, storage and retrieval, in areas where bespoke solutions had to be provided previously. The theoretical concepts illustrated in this dissertation are demonstrated through a working distributed implementation and deployment in several application scenarios.

*To my parents, John and Angela,*
*and to my dearest Stefanja*

# Preface

Except where otherwise stated in the text, this dissertation 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 I have submitted for a degree or diploma or any other qualification at any other university.

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

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

# Acknowledgements

The assistance and friendship of a number of people has been invaluable during my time at Cambridge.

First and foremost, my most heartfelt thanks are due to John Bates, for countless invaluable discussions, ideas, practical support and for being a constant motivating force that helped me see this research to conclusion.

I wish to thank my thesis supervisor, Jean Bacon, for providing me with the opportunity to work within the OPERA group, making me welcome, and providing continued patience and support throughout. Her experience and insight were of great assistance.

David Halls, Sheng Feng Li, Chrys Mallia, Ian Marshall and Scott Mitchell collaborated with me on a number of research projects. These collaborations helped me gain useful insight into a number of research areas that inspired the approach of this investigation. Giles Nelson and Mohamed Afshar provided much needed experience, encouragement and helpful suggestions that were to prove spot-on time and time again.

I wish to acknowledge the very helpful comments and suggestions from my four reviewers, who dedicated much time and effort to perusing various drafts of this manuscript; John Bates, Jean Bacon, John Hine, and Giles Nelson.

This work would not have been possible without the support of my parents, John and Angela Spiteri, who throughout my academic progression, were always a source of encouragement that made me strive to kept improving and embrace new challenges. I am deeply indebted to my partner, Stefanja Casha, for her love, emotional support, and for always being there. I particularly wish to thank her for never complaining during the long periods when my commitments in Cambridge kept us apart.

Finally, I am grateful to the Association of Commonwealth Universities and the Maltese Government for awarding me a Commonwealth Scholarship that supported me financially for three years.

# Publications

Aspects of the work described in this dissertation feature in the following publications;

John Bates, Mark D. Spiteri, David Halls and Jean Bacon, "Integrating Real-World and Computer-Supported Collaboration in the Presence of Mobility", *Proceedings of IEEE 7th International Workshops in Enabling Technologies: Infrastructure for Collaborative Enterprises*, Stanford, CA USA. June 1998 [*†]

John Bates, Jean Bacon, Ken Moody and Mark D. Spiteri, "Using Events for the Scalable Federation of Heterogeneous Components", *Proceedings of 8th ACM SIGOPS European Workshop*, Sintra, Portugal. September 1998

Mark D. Spiteri and John Bates, "An Architecture to support Storage and Retrieval of Events", *Proceedings of MIDDLEWARE 1998, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Lancaster, United Kingdom. September 1998

Ian Marshall, John Bates, Mark D. Spiteri, Chris Mallia and L. Velasco, "Active Management of Multi-Service Networks", *IEE Electronics and Communications Colloquium on "Control of next generation networks"*, London, United Kingdom. October 1999

Sheng Feng Li, Mark D. Spiteri, John Bates, and Andy Hopper, "Capturing and Indexing Computer-based Activities With Virtual Network Computing", to appear in *Proceedings of the ACM Symposium on Applied Computing*, Como, Italy. March 2000

Jean Bacon, Ken Moody, John Bates, Chaoying Ma, Andrew McNeil, Oliver Seidel and Mark D. Spiteri, "Generic Support for Asynchronous, Secure Distributed Applications", to appear in *IEEE Computing.*

[*] Awarded Best Paper of workshop.
[†] Awarded Best Paper Overall of conference.

# Contents

Chapter 1

# Introduction

In recent years, the communications paradigm of *event notification* has developed from a bespoke communications model in applications like graphical user interfaces to a comprehensive and feasible solution for dynamic software integration.

Event-driven and messaging infrastructures are emerging as the most flexible and feasible solution that enables rapid and dynamic integration of legacy and monolithic software applications into distributed systems. This is of major significance since more demanding customer service requirements are bringing about rigorous needs for comprehensive networked services. Event infrastructures also support deployment and evolution of traditionally difficult-to-build active systems such as large-scale collaborative environments and mobility aware architectures. However, unresolved issues like mobility, scalability, federation and persistence within these infrastructures hint at a requirement for more advanced services. The event notification paradigm is also applicable in emerging research areas like modelling of business information flow within organisations, as well as workplace empowering through identification and enhanced awareness of work practices relating to communication and interaction between individuals. In these areas, further developments require complex interpretation and correlation of event information, highlighting the need for an event storage and retrieval infrastructure that provides the required groundwork.

It is the thesis of this dissertation that the lack of a generic model for event representation and notification has restrained evolution within these event-driven applications. Furthermore, in order to empower existing applications and enable novel solutions, a crucial, and so-far-missing, service within event-driven systems is capture, persistent storage, and meaningful retrieval of the messaging information driving these environments. This claim is partly motivated by the wealth of information that can be gleaned through perusal of an event history. In addition, events represent indexing points into application sessions, and therefore an event history corresponds to a history of interaction.

This first chapter introduces the paradigm of event notification as a communications model, introduces the traditional and emerging areas of application of the paradigm, and examines the potential in retaining and reviewing event histories. After outlining the issues involved in designing a comprehensive event storage service, it then documents how this dissertation tackles these issues.

The chapter is organised as follows. Section 1.1 introduces the knowledge-gathering model behind the paradigm of event notification, while Section 1.2 lists the application domains where the paradigm is applicable in one of its many flavours. Section 1.3 introduces the notion that past activity in an event-driven system is portrayed in its event history, and hints at how this could be employed by an application to enable and enhance functionality. Section 1.4 then outlines an event repository service that addresses these issues. Section 1.5 describes the research issues involved in identifying the requirements relating to the storage and retrieval functionality required, and leading to the defining of an appropriate architectural design for an event repository. Section 1.6 concludes this introductory chapter by outlining the structure of the rest of this document.

## 1.1 Event notification

The concept of *event notification* is straightforward to introduce. Event notification is concerned with propagation of information. In a very simplified view of an information space, there are two distinct categories of entities. Some entities possess *useful* knowledge, either because they brought it about in the first instance (like a change in their internal state), or else because they found out about it somehow (by interacting with other entities, devices, or people). Other entities do not have this useful knowledge, and need to be *aware* of it in order to carry out their working tasks and obligations. The term 'useful knowledge' does not denote that information is universally useful. Knowledge that is very useful to an entity can be of no consequence whatsoever to another. These knowledge-requiring entities therefore need to obtain it from the knowledgeable entities. There are two ways for them to proceed:

- periodically, or even all the time, they have to communicate with the knowledgeable entities to see if they have any useful material, and obtain it from them when the latter have any, or
- they communicate to the knowledgeable entities some information about the knowledge that they are interested in, and the latter send it to them when they have it, or when that *event* occurs.

A useful analogy of these choices can be found in any modern email application. Consider a computer that is connected permanently to a network, and can receive email at all times. How can one tell when email has arrived? There are two ways to achieve this; you either look at the email application's inbox regularly (as in, for example, every few minutes), or else you can set up a notification action (like playing a sound) whenever email arrives.

This is analogous to an entity asking to be notified whenever an event of interest to it takes place. However, one often needs more information than that. An event is likely to have distinguishing *properties* or *attributes*. For example, receiving an email is a valid and useful event, but some attributes of that event are who is the email from, and what is it about. If one returns to the email notification example, one might wish to extend the notification mechanism to play a different sound according to who is the sender of the email, so that users can better tell when it is worth their stopping work to read email.

Event Notification is the embodiment of the second strategy, where an event is that occurrence of some useful knowledge. In an office environment, an event can be a door shutting, a telephone ringing, a user walking from one room to another, or the temperature in the meeting room becoming too hot. It can also be the act of someone logging onto a computer system, opening a software application, or editing a document (both real and digital). In a different world, a gaming world, it can reflect firing one's weapon in a first-person three-dimensional shooting game, whereas in a collaborative space it can denote a virtual user's interaction with a virtual object. Other representative examples are a change of state of a variable within a distributed program as it is debugged, or web server's load reaching a particular threshold. These few but varied examples give an indication of the vast nature of activity that occurs in the real and virtual (computer-based) worlds that can be of interest to someone, something, somewhere.

A crucial aspect of these event occurrences is that they can occur *at a time outside the control of the interested parties*. The entities interested in knowing when some relevant event occurs have no control over when it can happen. On the other hand, the *producers* or *sources* of events cannot know what, or how many, entities might be interested in the events they know about, and for how long. This latter point is brought about because one might only be interested in information for a length of time, as the interest might be brought about, or extinguished, by other incoming information. These two aspects clearly define event notification as a model of *asynchronous* communication, where entities communicate in order to exchange information, but do not directly control each other. Figure 1.1 illustrates this notion of asynchronous notification.

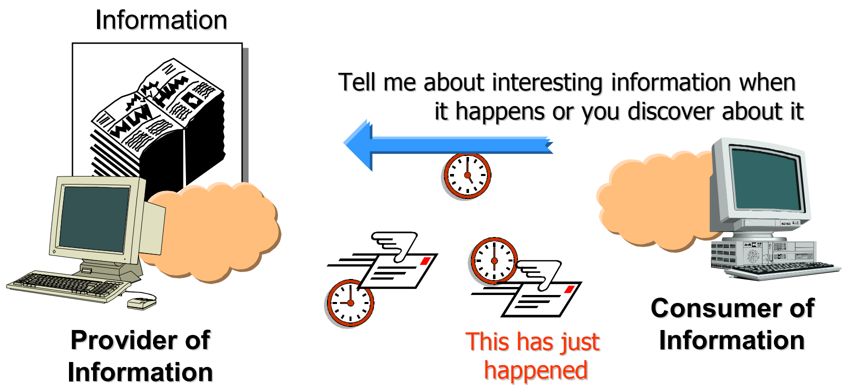

**Figure 1.1**
Asynchronous notification of information

## 1.2 Applications of event notification

The concepts behind event notification have found extensive application in several areas of computing. While its asynchronous model of communication can be very empowering, it is not universally applicable. In general, an event-driven system is composed of a number of independent (i.e. can exist and run independently of each other) and reactive components. This latter aspect is of major importance. In an event-driven system, execution of actions is carried out in a reactive manner; that is in response to external triggers. This differs considerably from other models of execution and composition that require entities to directly control and be aware of each other, and be *tightly coupled*. Through the independent reactive execution of its constituent components, the system needs to be able to achieve its design end-goals of service and aggregate functionality. These differ according to the application area.

There are also a number of variations of the event-based style. These variations impact on the structure, the behaviour and the performance of applications. A number of systems based around the event style have evolved as tailored solutions for particular problem domains, while others were designed from scratch as general-purpose infrastructures. While the terminology used in these systems varies, and the level of functionality provided in each differs greatly, the primary concepts underlying each system are similar. In each of these paradigms *events*, *messages*, *announcements*, *notifications*, *actions*, or *traces*, are used as the glue to integrate *programs*, *modules*, *tools*, *applications*, *processes*, *process groups*, *clients*, *objects*, *information objects*, *components*, *tasks*, *senders/recipients*, *agents*, *actors*, or *function hosts*. It is due to this breadth of jargon available that the above discussion employed conceptual terms like entities and knowledge.

Event-driven notification is particularly suitable:

- as an architectural style for building and rapidly composing large scalable distributed systems that can evolve as required by modern Internet-based dynamic environments.
- for enterprise application integration, where distinct (and sometimes legacy) applications, under common or different political and technical management, need to be integrated into a larger distributed system.
- within programming language environments, as a means of decoupling communications between concurrent objects and modules.
- for representing and enabling business information flow within an organisation. Knowledge and business events flow within a company and between companies. It makes sense to model this macro-level flow of information directly onto the supporting software system in terms of event notification.
- in describing micro-level work practices that often exhibit notification properties. In a physical workplace, individuals keep themselves up-to-date through interaction with other workers in a variety of ways. They then react to the information gained. By collecting data from the real and digital worlds, and

digitally representing this information, it is possible to enhance workplace awareness and optimise inter-worker communication and collaboration.

- for building distributed collaborative shared platforms, where multiple users interact upon a shared virtual space. This includes multi-user distributed games, virtual worlds, and computer-supported collaborative work (CSCW) environments.
- for monitoring and measurement-taking applications. The breadth of these is vast and diverse, from monitoring of distributed systems, centralised and distributed application debugging, to usability studies.
- for real-time environments that are mostly sensor-driven. In addition to the previous point, this covers control and management of physical environments like home-area networks, engine management systems, and telemetry data acquisition and management from vehicles and aircraft.
- within Active Databases. An active database monitors its storage operations and can carry out specific actions upon the occurrence of certain conditions pertaining to the state of the data it contains.
- for windowing systems that are usually composed of multiple control elements representing the various visible and virtual layers making up a graphical user interface. These generate miscellaneous events pertaining to mouse and keyboard activity within their scope, and these have to be passed around to other windowing or control elements that are interested in them.

The above list is not exclusive, and various applications overlap across the above loosely defined categories.

While this section has introduced the main areas where the event-based paradigm is relevant, Chapter 2 reviews these application domains in further detail.

## 1.3 The past, the present and the future

The paradigm of event notification is concerned with finding out about information when it occurs and as it occurs. It reflects an act on the future. An event-consuming entity is informed about changes or events pertaining to some knowledge that is deemed interesting to it (maybe because it registered an interest with the source of that information). By its very nature, this is restrictive, as it implies that there is no access to the *history of evolution* of that knowledge. In order to get to its current state, the item of interest will have gone through several permutations and changes over its lifetime. Its nature will have changed in various ways, which may itself be meaningful and interesting. Reviewing the history of occurrence of events and the evolution of their attributes over time can uncover particular patterns of behaviour that one can take advantage of.

Storage of event information, coupled with meaningful access to it, implies detailed knowledge of event history, much as event notification provides for knowledge of current information. As well as having auditing value, a history of events can enable reconstruction of state.

Event notification history ranging over a whole system can identify the flow of event information throughout a system, and recognize the causal relationships between generation of events at different locations. The distinct patterns in a workflow can be identified, bottlenecks and failure points located and acted upon. Past events earmark crucial checkpoints in the execution of a system. Furthermore, by analysing the consumption patterns of event clients one can take steps to optimise the delivery of future events to those clients so that the information is better 'contextualised' for them. Future service can be improved.

Past event information can be queried, browsed, analysed, and even replayed. In short, knowledge of past events can be as useful as knowledge of live events.

## 1.4 Storage and retrieval of event information

In order to enhance the existing functionality of event-driven systems and enable the creation of novel applications, this dissertation identifies a new requirement. The events used to glue together these application entities can be represented through a generic model, stored, used for querying and replay, and as a basis for higher-level services. The motivation for storing events is that events represent indexing points into application sessions and correspond to a history of interaction.

In order to achieve this, this dissertation presents an *event notification and storage infrastructure* that can capture and store events, enable querying on event histories, as well as enable *injection* of events back into distributed application components to simulate replay of sequences of activity. Changes in the state of a distributed system can be captured as events, and replayed or reviewed at a later stage, supporting fault-tolerance, systems management, disconnected operation and mobility. Using a generic object-oriented model for events, events are notified around a distributed system and retained within *event repositories*. The architecture of an event repository delivers powerful search and retrieval facilities, enabling extraction of behaviour patterns, searching for simple and composite occurrences, and replay of intervals of stored sequences. This addresses business requirements in several industries (e.g. finance, travel, news, retail and manufacturing) to locate temporal patterns of activity, as well as provide a viable alternative to past system-specific solutions in areas like logging of collaboration and 'human memory prosthesis'-based tools. In the latter, information is gathered about the events that occur in the physical and virtual (digital) working environment of users. While past implementations [LN93, LBC+94] have provided automated diaries that could be browsed to assist recollection of activity, coupling with an event repository enables thorough querying and analysis capabilities. Event storage also enables novel applications such as visualisation and analysis of user mobility. In an example of this, the information pertaining to a history of physical movement of people in some environment, like an airport or an oilrig, can be employed to generate three-dimensional animated replays of movement. This can be applied to improving security and safety in buildings.

The repository offers a selective store-and-forward functionality that enables messaging environments to scale and provide enhanced brokering and federation services. The architecture presented co-exists with and enhances event-based systems while providing a cheaper, more flexible solution in areas where previously custom monolithic designs had to be provided. In addition, bespoke niche queries can be defined in support of specialised legacy and novel applications.

Other application areas identified as beneficiaries of this architecture are systems monitoring for distributed debugging, tracking of phone data for telecommunications fraud detection, and acquisition of marketing information for Internet commerce. Within some scenarios, it is desirable to be able to use information from all these different sources together, hence creating a better and more fine-grained virtual picture of the human and computer activities monitored.

Chapter 2 will review a wide range of application areas where an event repository service can be deployed. In these areas, it either serves as a viable alternative to bespoke solutions, or acts as a novel enhancement, or provides the necessary groundwork for emerging requirements.

It could be argued that a conventional database (relational, object-oriented or temporal) would be suitable for carrying out the same function as the event repository being proposed. However, a properly designed event repository is significantly more powerful than its conventional database counterparts because:

- an event repository can efficiently perform functions that in conventional database systems must be encoded in applications, *e.g. temporal indexing, cross-type event session organisation, event schema evolution, template matching, and querying by sequential instance comparison*
- an event repository suggests and facilitates applications beyond the scope of a conventional database, *e.g. replay of event sessions*
- an event repository can perform tasks that require special purpose subsystems in a conventional database, *e.g. temporal indexing, retrieval and replay, archiving*
- the highly specialised storage and retrieval required of event information justifies a tailored storage engine and does away with the requirement for most heavyweight database functionality, *e.g. read-only historical access, very high-performance write append, small event records, temporal indices, searching and retrieval by sequential correlation*

This issue will be discussed at further length in Chapters 5 and 6.

## 1.5 Research issues

The main issues investigated by this research, and discussed within this dissertation are:

- *A model for representing generic event instances.*
  An event storage service can be used in a plethora of application domains where the nature of the event or messaging information varies. A suitable event model

is one that is simple enough to deploy generically, but can be easily extended if required to accommodate the requirements of specific applications. It has to enable the definition of events that can represent the wide variety of activities and occurrences that tie together event-driven systems, while supporting adequate granularity of searching for, and filtering of, interesting data. A generic model of event representation is therefore defined.

- *An architecture for seamless integration of the functionality and services of an event storage repository with different environments.*
  Event capture in distributed environments is more complicated than in a centralised system. A distributed system will consist of a number of processes executing on different machines and communicating via message passing. Centralised detection of event instances is not feasible in medium to large scale distributed systems, because of the possible volume of event instances to be monitored, delays introduced by the network transmission, and unsynchronised system clocks across different machines. The differing distribution details of various applications, as well as emerging requirements for loosely coupled distributed systems, were investigated and taken into consideration while designing a scalable solution for dynamic event propagation, capture and storage.
- *A flexible storage service that embeds a high-performance storage paradigm tailored for the particular nature of event data.*
  An event repository must be flexible in its design, so that it can be customised for environments with constraints on memory, storage and processor availability. In addition, a range of interfacing issues needs to be considered. Although event instances could be mapped to relational records or tuples, or object instances, and a relational or object-oriented relational database be used for storage, there are several reasons to tailor the underlying storage engine. Event instances tend to be relatively small structures, and since they constitute a history, are temporally ordered and immutable. The storage engine can thus be greatly optimised to address heavy volumes of incoming event streams and avoid the unnecessary overhead of conventional general-purpose databases. Databases can have very high costs associated with operations like insertion, deletion, as well as services like transaction management and data rollback support, facilities that are not as relevant for event storage. The data as stored must be representative of the causal and temporal separation of the events as they were received in order to support the deployment of a temporal retrieval interface over it.
- *A powerful interface for retrieval and replay of information from event stores.*
  Although general-purpose query languages like **SQL** [ITI98] or **OQL** [CB97] could be employed to retrieve information, there is much to be gained by designing a query interface optimised for the temporal nature of the information stored. Queries over temporal data and relating to time-intervals, as are event instances, involve references to time which benefit from purposely-defined temporal operators. The querying interface must permit expressing queries over sets of different event objects or their aggregate types (composite events). This dissertation presents a query interface that offers a superset of **OQL** functionality. Therefore, while providing a standard database interface for

applications, a repository can also support client applications that require analysis and interpretation of event histories. This interface must integrate smoothly with the event-driven environment, allowing seamless access to past event data and its feeding back into the system.

## 1.6 Dissertation outline

This dissertation is organised as follows.

Chapter 2 addresses the motivating factors behind the thesis of this dissertation. It starts by describes the primary application domains where the concept of 'event' persistence has been applied or is applicable. It then describes how the application of event storage can enhance the functionality of these applications as well as lay the architectural groundwork for more complex features. In doing so, it also reviews research from an application perspective that is directly relevant to or has influenced this dissertation.

Chapter 3 focuses on related existing event-driven and messaging frameworks. It reviews several research and commercial event-based approaches and describes their main features. Usage of event persistence within these systems is highlighted.

Chapter 4 introduces HERALD, a new event-notification communications infrastructure that provides novel services for federation and dynamic application construction and evolution. This chapter also introduces a general-purpose event model. HERALD embeds and actively employs event storage, thus demonstrating how event storage can be integrated within an event system to provide a comprehensive event-driven framework.

Based on the motivating factors identified in Chapters 2 and 3, Chapter 5 then identifies the information retrieval and replay facilities required of an event repository. It addresses these requirements by introducing an event query language, TEQL, designed to offer the retrieval power of an industry-standard language while supporting a comprehensive temporal formalism for events.

The architecture of a general-purpose event repository is presented in Chapter 6. This chapter describes a flexible design that can be customised to address the specific requirements of the variety of application domains event storage is applicable within, and discusses the issues involved in designing a storage sub-system tailored for the particular nature of the data being handled.

Chapter 7 outlines a prototype implementation of the technologies discussed in this dissertation and discusses deployment issues for event repositories. It then describes a number of applications built around, or employing, the capability of an event notification and storage infrastructure. The diversity in application domains and environments these applications are representative of highlights the flexibility of the design presented.

Chapter 8 discusses the solutions presented in this dissertation. It gives a summary of the research, reviews insights gained in the course of this investigation, and suggests directions for future research.

Chapter 9 concludes this dissertation, and highlights the main contributions.

Chapter 2

# Applications and Motivations

This chapter *motivates* the premise of this dissertation, that a crucial service within event-driven environments, is capture, storage and retrieval of the event information driving these environments. A major motivation of this work is that it has been illustrated by application-specific implementations in a number of research areas that event histories are required for providing advanced functionality. However, in these implementations, the lack of generic models for event representation, notification and storage has severely constrained the extent of the functionality that could be supported.

Section 2.1 introduces the application domains where event notification has found application, or is emerging as an applicable solution. It also comments on the origins of the event-driven paradigm. The following sections then address the most representative applications in the categories identified in Section 2.1, where, within each domain, *event storage is applicable to enhance functionality*, or otherwise *is required groundwork for envisaged future services*. Realistic developments that can be envisaged as resulting from building over an event storage service are highlighted.

Section 2.2 introduces the use of event notification in application integration and construction of distributed systems. Specific event solutions are not examined in this section, as this is done in greater depth in the related work overview given by Chapter 3. After examining the potential of event storage, it considers the area of fault-tolerant computing and distributed debugging, where capture of event histories has been applied.

In Section 2.3 the emerging area of workplace awareness is described. Future research directions in this area are explored, and it is argued that these developments need a comprehensive event storage service as a crucial starting point.

Event notification and storage in computer-supported cooperative work are examined in Section 2.4, while Section 2.5 looks at a number of other interesting application areas, like visualisation of mobility, the networked home, and graphical user interface evaluation.

In order to avoid confusion this document uses the phrase *event storage* predominantly to imply *event logging*, *event tracing* and *event capture*; terms that are often employed in the related literature. Except where specified otherwise, the phrase is also taken to denote not only the act of event detection and storage, but also an aggregate feature-set consisting also of event interpretation, transformation, retrieval and replay.

## 2.1 Applications of event notification

As introduced in Chapter 1, there are several areas of computer science where the event notification paradigm has either found application, or has emerged as an applicable solution to address evolving requirements.

In an *active* environment, an application is composed of a number of software entities that react to each other's activities rather than directly control each other. An entity therefore *triggers* activity in another one by sending it a message in some structured format known to both that details the event. Such an entity is known in this document as a *source* of events, i.e. an entity that, further to some user input or device monitoring, can generate these event triggers. In other literature, such entities are sometimes called servers, but this association with the client/server paradigm (see Section 3.1.1) can be misleading. Sources either maintain information about the other entities that are interested in their events (the *clients*) and dispatch them directly to them, or in a variation on the model (employed by several event-based systems), pass them on to a centralised or local notification service. This module is then where the data pertaining to interested third parties is retained and from where the event messages are dispatched.

This model of communication is based around the premise that the client would first have expressed its interest, with either the notification server or directly with the source, through a registration or subscription of some sort. However, in several applications where the clients and sources are static in number and nature, the target of notification can be hard-coded into the application logic. In this case, communication can be said to have reverted to the messaging model. This more-general method of communication is widely used in application integration (see Sections 2.2 and 3.4).

The discussion above and in Chapter 1 has used the conceptual and vague term *entities* on purpose, for two reasons: firstly, because there is no consistent vocabulary for discussing event systems; secondly, because their actual embodiment depends on the application domain under consideration. From a software perspective, an entity can be a module within a process, a process within a multi-threaded application, an independent component within a distributed system, or even a whole system that is communicating with another system over a wide-area network.

As listed in Section 1.2, event-driven notification is at the heart of countless software applications. These fall into one or more of the categories below:

- Composing and building large scalable distributed systems; Enterprise application integration; Concurrent/parallel programming languages; Modelling of information flow; Distributed debugging and fault-tolerant distributed systems (see Section 2.2)
- Modelling and supporting work practices relating to communication in between individuals (see Section 2.3)
- Building computer supported cooperative applications and collaborative shared platforms (see Section 2.4)
- Applications based on monitoring and measurement-taking; Real-time applications, i.e. applications that have to constantly act on data received from physical or virtual sensors; Active databases; Windowing systems and graphical user interfaces (see Section 2.5).

The above grouping reflects the organisation of the discussion in the remainder of this chapter.

Often, the major applications representative of the above categories evolved into using such a reactive model of inter-component interaction independently and in response to emerging requirements. That is, this was not due to the application of any formal software engineering architectural style. In fact, the notion of an event-driven style has been formalised (see [BCTW96] for one such formal model) primarily as the result of examining the concepts in common amongst reactive applications. The implication of this is that there is a plethora of bespoke and incompatible designs and implementations of event-driven systems. However, their common basis makes it possible to reason about an event-driven system in a generic sense. Likewise, this allows one to reason about the knowledge that can be gleaned through study of event histories. Based on this premise, this dissertation proposes a generic event storage architecture that can be deployed and employed in different instantiations of the event-driven paradigm.

The following sections will now describe in further detail some applications that are representative of the above categories. The discussion will look at the potential for retaining event histories in each application, and describe functionality enabled by event storage.

## 2.2 Application integration and building of distributed systems

A major motivation of this work is to provide a generic and scalable way for integration of distributed components. Currently this is needed in important business applications such as determining business information flow. Current techniques for integrating components lack interoperability and thus hamper scalability.

The deployment of distributed systems is a major area of application of event-driven systems at present on a commercial basis. It is also an umbrella category, in that conceptually it encompasses most of the other categories. While this section discusses the issues involved in application integration, actual event-driven solutions and the messaging systems they are often built from (Message-Oriented Middleware) are reviewed in Chapter 3. Most of the solutions applicable to application integration and construction of distributed systems are general-purpose in nature and are therefore representative of the bespoke solutions that are deployed in other areas, like those discussed in Sections 2.3 – 2.5.

### 2.2.1 Background requirements

The Internet's acceptance by the commercial world has created a global and standard communication infrastructure never available before. Distribution of one's software architecture across multiple networked platforms and systems is often a necessity dictated by requirements of customer service, management resources, or simply utilisation of the best existing, and rapidly evolving, technology. In most cases, re-design and re-implementation of the components required to form part of the new integrated system are not viable, if only because time-to-market periods have become so short. Alternatively, due to their distinct administration and management, it may be technologically impossible, or politically unacceptable, to modify existing systems extensively in order to make them work together.

The other primary issue with application integration is the dynamic nature of software systems. Few, if any, systems are not altered or adapted to changing market requirements during their lifetime. In some markets, like finance and the travel industry, the electronic information being traded and presented has its origin from multiple sources managed by independent companies. These are often faced with dynamically evolving diverse client and market requirements requiring continuous updating of their software systems. However, as such modifications may break the information chains of other companies, much duplication and complexity often needs to be introduced.

### 2.2.2 Event-based integration

Approaches to distributed integration have ranged from *loose*, in which the components have little or no knowledge of one another, to *tight*, in which modules require comprehensive information about each other. The latter requires knowledge of interfaces, communication protocols and data structures rarely available other than within a co-ordinated design and implementation project, or within environments where strict industry standards are available and adhered to (see Section 3.1.2 for further detail). *Loose integration* is therefore more suited to the varied and dynamic nature of a typical distributed system's components, as it helps reduce the impact on a system when modules are added or changed.

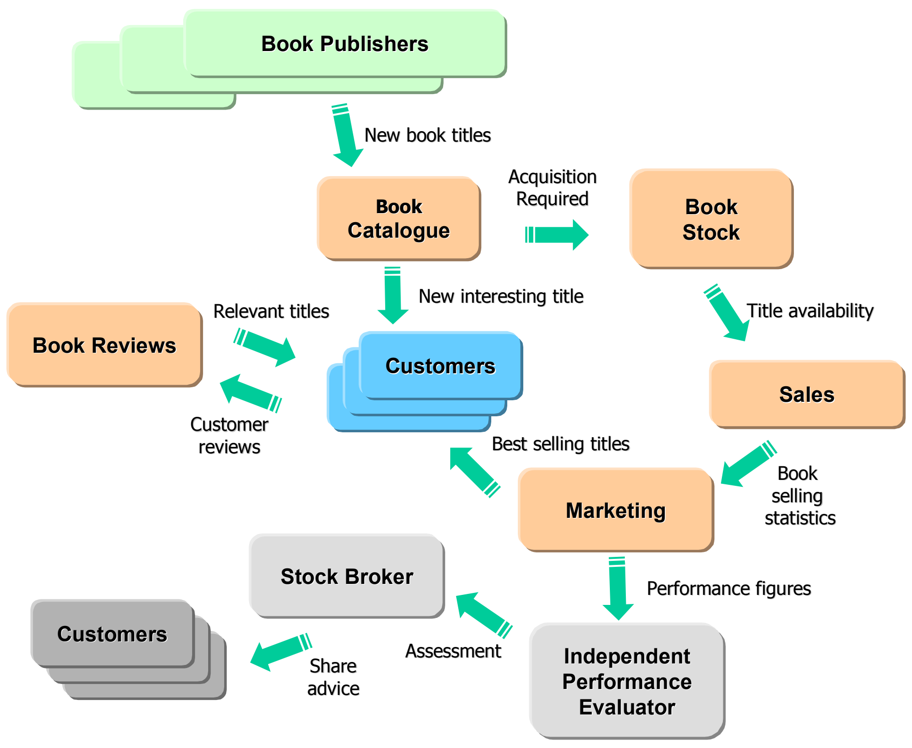

**Figure 2.1**
Event flow in an e-commerce based book-retailing operation

*Event-based integration*, in which distributed modules interact by announcing and responding to event occurrences (or messages), is perhaps the most prevalent loose integration approach. In what is also termed *publish-subscribe*, the modules and components to be integrated are turned into sources and consumers of events through their interfacing with an appropriate messaging or event-driven middleware. In most scenarios, event information can map directly to the main activities of a business. Examples of important events, depending on the industry, are movement of parcels, the ordering, purchase and sale of goods and services, changes in prices or conditions. Enterprises need to be responsive to these events, often in real-time. Figure 2.1 illustrates how asynchronous events can be used to integrate modules within a book-selling operation, as well as link it to external entities like publishers and independent financial assessors.

Chapter 3 analyses the issues involved in building distributed systems in some detail, and addresses the advantages and shortcomings of alternative distributed middleware solutions. It also describes the highlights of some key event systems and commercial messaging products, amongst which general-purpose solutions like the **Cambridge Event Model** [BBHM95, BBMS98], **ECO** [SCT95], **Yeast** [KR95], **FIELD** [Rei90], **Polylith** [Pur94], the **CORBA Event Service** and **Notification Service** specifications [OMG97, OMG98a], as well as commercial systems from IBM [IBM99], Oracle [Ora99], and Talarian [Tal98] amongst others.

### 2.2.3 Programming environments

Event notification has also found application in concurrent programming language environments, both in pure parallel languages, and in languages that support multi-threading. Several languages allow execution to be split along several concurrent threads or processes. Synchronisation and communication between these concurrent paths is then carried out through protected shared data or through message passing. Some [Reu80, Sha89, SCT95] extend the intra-process messaging model into a fully-fledged event model, where events are delivered by the language's runtime system in an asynchronous fashion from process to process according to the requests of the processes. These systems are akin in functionality to a subset of distributed event models, and will therefore not be addressed explicitly any further in this chapter.

### 2.2.4 Applying event storage to distributed systems

In distributed systems, the events propagated in between the distributed components are the *systems glue* that ties together the otherwise distinct components. As events propagate throughout a distributed system, they trigger actions in components, some of which generate other events of interest to other components. This flow of events and actions often corresponds to the flow of business information in an organisation or in between organisations. Capturing details of the event streams, whether at their source, or at distributed nodes, or even at a centralised location, can enable one to identify the main paths of execution throughout a system, and locate inefficiencies and possible failure points. At present, this interpretative functionality is available using *workflow systems* [AAEM97, CHRW98, GHS95]. However, the information that a workflow system allows analysis of needs to be entered by a user, and is not obtained in an automated fashion directly from the system. Since event flows often map directly to business flows, it is envisaged that event histories could be used to feed a workflow application with little or no user intervention. Such an analysis would be very useful in a loosely coupled and dynamically evolving distributed system. Event histories can be used to modify the service provided and the business flow itself.

The following scenarios illustrate functionality that can be enabled by building on top of an event repository service in the context of systems integration:

- *Auditing* of event generation at event sources, event transformation and delivery, as well as event consumption at clients.
- *Identification* of periodically repeating events and attribute evolution for event instances of the same type.
- *Analysis* of event registration, notification and consumption at client components. This can yield information on how different event notifications influence a client's execution with respect to their sequence and over time. This can allow one to determine when information is most useful for a client and modify the delivery or contents of future information to contextualise it better.

- *Tracking* of repeating patterns of activity covering one or more event types over a number of event sources and clients.
- *Checkpointing* of execution. The history of event interaction between event components is analogous to a history of evolution of state in the system. Supporting the capture and retrieval of events is analogous to recording checkpoints (for rollback purposes) in a database system. The main points of interest in an activity can be reconstructed later using these temporal indices. Among other uses, this can support the construction of fault-tolerant distributed systems. See the next section for further discussion of this point.

### 2.2.5 Debugging distributed systems

The last bullet point of Section 2.2.4 introduces two research areas; *fault-tolerant distributed systems* and *distributed debugging*. These research areas are interesting because of the problem of *non-determinism* that arises in them. The independent execution of multiple processes communicating with asynchronous protocols introduces a large degree of non-determinism in a distributed application. This makes it difficult to implement debugging functions such as distributed breakpoints in an efficient manner. Several treatments of these aspects of distributed debugging have appeared in the literature (see [PN93] for an overview). The most widely acknowledged way of addressing this is to wrap event monitors around the distributed modules of the application being monitored or debugged, and then employ an event propagation mechanism to propagate the event information to where it can be stored. LeBlanc et al. [LR85] and Bates [Bat95] illustrate how heterogeneous distributed systems can be debugged using event-based models of behaviour.

Several authors [For91, HS92, LIT91] have examined event detection and capture in the context of fault-tolerant computing and distributed debugging. Logs are maintained of event 'traces', where these traces represent low-level system activities like processor communication. In particular, [HS92] examines how events generated to reflect activity in a system can be coalesced into a smaller number to reduce the size of event logs, as well as avoid redundant events. The relationships between event log entries, system workload and system configuration have received attention by several researchers, amongst which [Han88, Tsa83, IYS86, LS90]. These provide techniques as to how event logs can be examined for determining trends in application execution.

By capturing all events pertaining to remote-procedure calls (RPC) and other communication primitives going into a process, and retaining them in an event repository, during a debugging session one can feed back the event data (or use it to reconstruct the primitives) into one or more of the distributed modules. This enables selective replay of the application where the user has control over the data being input into the module and can test it in isolation.

Methods to reproduce the execution behaviour of programs comprised of loosely coupled processes that communicate using messages typically require that the contents of each message be recorded in an event log as it is received [CW82, Smi84, TA87, Wit88]. The programmer can either review the events in the log in an attempt to isolate errors, or the events can used as input to replay the execution of a process in isolation. In order to address variations in scheduling and message latency during multiple executions, the order in which messages are delivered can also be traced. Since parallel programs are long-running, providing fast response to debugging queries requires *incremental replay*, where re-execution is started from intermediate states instead of from the beginning. To support incremental replay, processes must be checkpointed periodically and the contents of some messages captured. *Adaptive event-message logging* can be employed in order to reduce the volume of messages that need to be retained [NX93].

Checkpointing and message logging has also been studied in the context of fault-tolerant computing [JZ88, SY85, WF92]. In [LIT91], statistical techniques are applied to automatically generated event logs from fault-tolerant systems in order to measure dependability.

All these systems employ proprietary mechanisms for event propagation that are very specific to the particular application, and provide highly focused retrieval interfaces to the logs of event traces retained. More than lack of functionality, this has contributed to their lack of uptake in the commercial world. The generic model for event representation and the event storage service proposed address these issues, and allow wider scope in event trace analysis.

## 2.3 Awareness of working practices

An event storage architecture is required in order to provide much needed homogeneity and to support development of the novel analysis and interpretation needed within this emerging research area.

The study of work practices in organisations has only recently gained research attention. Ethnographic workplace studies carried out by the Xerox Research Centre Europe [BRS+94] and others [Tan91, KFRC93, WFD94, Fro95] in businesses of various kinds as well as of institutions like hospitals, have revealed that the way in which individuals run their day-to-day duties is often reactive and exhibits notification properties analogous to an event-driven model. Apart from organised venues of interactive sharing of information, like organised formal discussion meetings, most work activities are in fact coordinated through informal interaction between users, unscheduled notification, or discovery of relevant information.

For example, a business manager may want to keep himself up-to-date by requesting that specific intelligence is sent his way immediately by his staff. He does not wish to be informed of everything, but does want to know when certain events take place, or when certain sequences of activities happen. From his perspective, this information can be passed to him at any time, as seen fit by his staff. When he does find out about matters of interest he takes some action or starts some work process in response. This is analogous to carrying out a subscription of interest in the occurrences of a primitive or composite event in an event notification environment, and then taking some meaningful action on notifications of the event. In most routine organisational work, it is discovery of information that is the trigger for most work activity. Information can be received at any time through various means, e.g. receiving email, phone calls, people visiting, incoming paper correspondence, or the publication of a relevant article in a magazine, newspaper or web site.

Likewise, knowing about a colleague's status and being *aware* of their current activities is very useful for coordinating activity. This information, on the other hand, is rarely notified to users in the real world. One does not know that a file has changed, a colleague has arrived, a new directory created, and so on, unless one explicitly thinks to *check*. In short, few office jobs can be carried out in an isolated information space, i.e. one with no means of interacting with colleagues and external news sources [Whi96, WSKS97]. These sources of information are like event sources, and their occurrence is in itself an event, much as their contents are analogous to the attributes of an event.

The challenge for these studies is to achieve the explication of work practices in terms of the structure of the activities and interactions involved in work. These can be embodied in technology intended to support work practices. Systems like **Khronika** [Lov91], and the **TickerTape** built on the **Elvin** notification service [FMK+99], attempt to provide users with awareness of the activities or status of others through a continuous stream of events that describe user activities. They collect information pertaining to the real world in a variety of ways, by monitoring physical location, email notification and filtering, schedule and booking of resource monitoring (like booking a room for a meeting), and through file-watching and web-watching. These latter event generators monitor changes to the network file stores and to internal and external web pages, and dispatch or broadcast notifications. While not directly attempting to support awareness, **TeleNotes** [WSKS97] creates a shared context for communication where users can follow conversations through 'sticky' piles that characterize the temporal progression of the conversation.

A wealth of information can be collected from a working environment and used to deduce the nature and context of a user's activities. In addition to the events listed above, other noteworthy examples are; monitoring of workstation logging in and off, monitoring of execution of applications or opening of documents for reading and editing, creation of documents, browsing of web pages, composing and sending email, access to newsgroups, engaging in shared online collaboration, making phone calls and sending faxes, printing documents, etc. Even interaction with devices like the communal coffee machine can be interesting, and actually feature as an application in experiments [ATT99]. In themselves, these isolated events are of little consequence, but in aggregate, and in the context provided by their causal and temporal relationships to each other, they define a picture of activity and interaction.

The above point can be summarized as follows: *acquiring awareness of individuals' activities and making that awareness widely available*. If retained and interpreted properly, this awareness can help in identifying individual knowledge and expertise, and at the same time assist in its distribution throughout the organisation. Duplication of work, particularly as concerns seeking of solutions to problems, can be avoided. When associated with appropriate privacy policies, this can dramatically increase efficiency in an organisation. Some attempts to identify and make use of expertise have been carried out in controlled closed environments. Building on the newsgroup concept, **Beehive** [Abu99] is a commercial service that, through email and the Web, enables users to locate, contact and interact with others who have similar professional interests and want to share their individual knowledge and experience with one another. It monitors and scans discussion forums on a number of web sites to determine potential experts in any area by studying their replies to other users' questions. As its knowledge base grows and evolves, it attempts to pass on questions to the 'experts' most likely to have the expertise required to answer them.

### 2.3.1 Enabling functionality through event histories

It is clear that while live notification is very useful in this domain, major benefits can only be achieved through logging and analysis of the event information collected. A history of activity is the starting point that may be used:

- *for providing a memory recollection tool or diary of events,*
  In [LN93, LBC+94], Lamming et al. define the concept of a *human memory prosthesis*, a tool for assisting memory recollection. It is well known that human memory relies heavily on context for recollection. One might forget the details of a particular activity they undertook, but they are likely to remember details about its temporal relationship to other activities that were being undertaken at the same time, or occurred earlier or later. Events pertaining to an individual can be captured and used by that individual in a diary-like manner (see Section 7.3). Applying the underlying concepts of memory prosthesis over a generic event model and transport like HERALD (see Chapter 4) would provide knowledge acquisition of a finer granularity, due to the increased variety of events pertaining to the real and virtual worlds that users work with. When coupled with the event-storage and query service described in this dissertation, the resulting application would enable users to have expressive querying lacking in previous solutions.
- *for determination of work coordination and work flow,*
  Events pertaining to news distribution and users' actions can be retained and studied in order to determine how work is being coordinated in between individuals. This can yield very useful information on what are the most important incoming events that trigger activity, their sources, and the sequences of activities and other events that they cause to be generated. Insight can therefore be gleaned on the flow of work in a reactive environment, which by its seemingly unpredictable nature is hard to model. There exists some understanding on how to generate social process models from event traces [RN96].
- *for modelling an individual's role and execution of duties.*
  High turnover of staff in the computing industry (where individuals tend to have unique expertise and experience) is of major detriment to successful conclusion of research or development projects. When an individual leaves, the person taking on their role often ends up starting with a considerable set of documents

pertaining to the project the departing person was working on. The time required to browse, get acquainted with, and identify crucial data, through what may be mostly useless and irrelevant documentation, is both extensive and expensive. Furthermore, the new person has to determine how the documentation was arrived at and the best use it can be put to. They also have to figure out how the departing person executed their role with regards to their colleagues and recognize upon what information input or feedback did they carry out and develop their work, and produce deliverables. By acquiring and retaining detailed information on an individual's activities, and introducing the concept of a history of use of all documents by different individuals, it is envisaged that it may be possible, through automated and manual study of the history of activity created, to create some meaningful model of the departing individual's working duties. This belief is reinforced by the observation that 53% of workplace interaction involves a document [WFD94]. Indexing, cross-relating and retrieving email threads that relate to documents would also provide a temporal and personal context for the evolution of work surrounding those documents [WS96]. By manipulating and appropriately presenting this information, a new person can be assisted in getting up to speed quickly, since they can more easily determine the major characteristics of the role. This scenario as envisaged is illustrated in Figure 2.2. Research on the issues involved in carrying out such an analysis is currently ongoing at Xerox Research Centre Europe [Ben99].

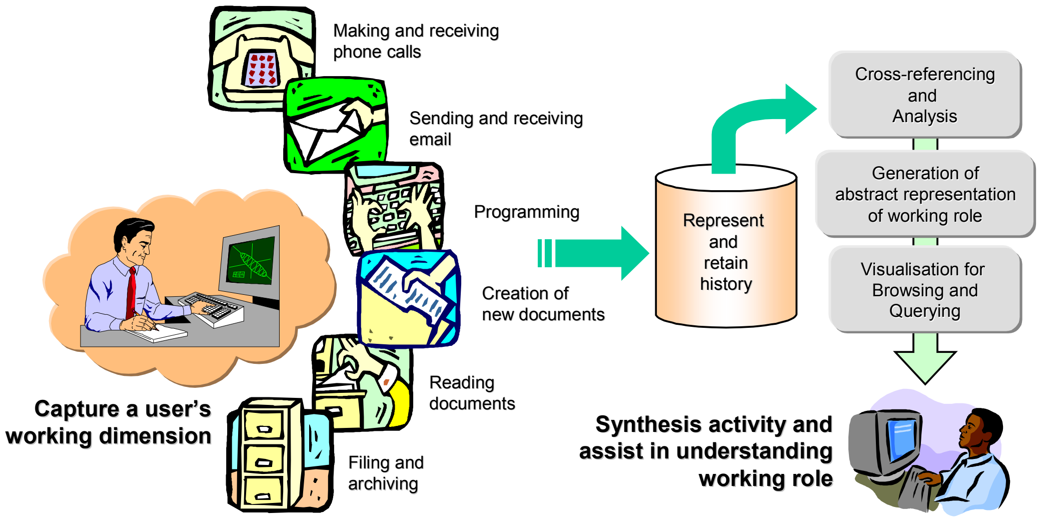
Making and receiving phone calls
Sending and receiving email
Programming
Creation of new documents
Reading documents
Filing and archiving
Capture a user's working dimension
Represent and retain history
Cross-referencing and Analysis
Generation of abstract representation of working role
Visualisation for Browsing and Querying
Synthesis activity and assist in understanding working role

**Figure 2.2**
Acquiring awareness of activities through events and keeping a history

## 2.3.2 Uses of event logging in the literature

Some aspects of the above have been attempted in various bespoke (and often limited in scope) experimental environments.

In their investigation, Lamming et al. [LN93, LBC+94] collected information pertaining to users' activities, like details of their location together with extensive video footage of various activities they undertook. After the records captured were textually annotated (through user intervention or through limited automated deduction), users could carry out limited searches on the *diary of activity* thus created, as well as locate the video footage relevant to the episode they wanted to recollect or retrieve information on. The non-extendible event model employed to model activities, and the restricted querying capabilities available, constrained the usability of the tool created.

In the **Where-Were-We** [MH93] project, workgroups were allowed to carry out playback of the video record as it is being taken, thus aiding recollection in case of distractions or the need to clarify or review previously discussed points. Building on this research, the **Coral** suite of tools [MHJ+95] supports real-time capture and subsequent access to informal collaborative activities. Manual annotation in real-time during the activity is required, this then being associated with automatically logged events like drawing on the shared whiteboard tool. The meeting can then be replayed from any instant.

Automated annotation of video (as mentioned above) would greatly assist the above indexing techniques. [GGR94] segments continuous audio and video into natural units and relates these to discrete events from the multimedia application, such as user interaction, control events, and data content. The latter are obtained by keeping a record of the most significant events sent to the **X Windows** server windowing software running on the participant's machines.

Another approach is employed by **CECED** [CLFS93]. This tool is intended to aid collaborative work in engineering design by capturing the history of the informal phase of the specification and design process. In this case, audio records are stored together with a log of the design traces, and system-level data like **X** events. This allows subsequent replay of design traces. Similarly, Bellcore's **STREAMS** project [CH94] focuses on making and accessing recordings of technical presentations.

In the above representative investigations, event logging is used to capture activity, but the logging capabilities are tailored to the application, and limited in functionality. No classification of events is carried out, and this makes the use of events specific to the application domain in question. The lack of a comprehensive event model implies that the system cannot evolve to take on new event types without requiring modification of the distributed components. It is not possible to abstract event types of similar semantic meaning into a collective type, and events cannot be composed into higher-level constructs. Retrieval is limited by the lack of a dedicated storage sub-system and is often constrained to **SQL**-like searches. Users are only able to browse through the activities undertaken, as annotated or recognised by the system. In order to address the storage scenarios outlined above, an appropriate query interface is required that has the capability of exploiting the temporal and causal relationships between event instances.

## 2.4 Computer-supported cooperative work

Some of the tools derived from studies of working practices between individuals (Section 2.3) can be termed *Computer Supported Cooperative Work* (CSCW) tools. It is proposed herein that by using events to communicate between and drive interaction within such tools, interoperability, flexibility and extensibility is enhanced massively. Communication mechanisms within current tools are often hard-wired (e.g. 2 distributed whiteboards communicating using a specific whiteboard protocol) thus hampering flexibility and extensibility. In addition, a new requirement outlined herein is the capture of the history of collaboration.

CSCW technologies enable multiple users to collaborate on some task. They often achieve this by applying the notion of a shared space where multiple users may interact on some common state. This may range from a simple common drawing surface, to a shared document that may be edited by several users concurrently, to complex three-dimensional virtual worlds [IKG92, GRWB92].

A crucial feature of collaborative interfaces is *feedthrough*, that is the ability of one person to see the effects of another's actions. Technically, there are two requirements that need to be addressed, firstly to access and update shared data, and secondly to know when that data has been updated. The latter requires notification mechanisms. Even if data is stored and accessed rapidly from a central location, it is of no use unless client programs know that it has changed and users' screens are updated accordingly. A notification mechanism fulfils this role, telling programs and users that changes have taken place. Without notification, users may eventually see that changes have occurred but at a timescale and pace that often are not acceptable for the task at hand.

Event propagation is a good model for describing instantaneous input in user interfaces and collaborative systems (like key-strokes, mouse clicks and network messages). However, collaborative systems emphasise the notion of shared data, where the event-based model fits less well. Shared data persists – it does not happen at a particular moment. To address this, a collection of semi-formal and formal techniques known as *status-event analysis* can be employed to define a shared conceptual framework that includes aspects of both events and data state [DFAB98]. This has been employed to model the complex behaviour of shared windowing elements in collaborative applications [DA96a], and has brought about an understanding of the delays in user interfaces and collaborative systems [DA96b]. [RDR98] investigates the design space for notification servers in CSCW applications and documents status-event analysis; i.e. the various ways in which event notification can be coupled with other communication mechanisms to bring together both event propagation and shared data.

### Capturing the history of collaboration

In addition to the scenarios enabled by storage of events outlined in Section 2.3, there is another aspect to CSCW tools that is empowered through use of event histories. Because the majority of input events in a cooperative tool are sourced from an entity external to the system, the user, most events are not directly causally related. For example, when a user draws a line on a shared white-board, that event is propagated to all the other shared whiteboards, where its interpretation involves drawing a corresponding line. Therefore, the action that triggered the event came from outside the system, and is not itself the result of some automated computation in response to some other event. This implies that capturing all the triggered events, and replaying them is analogous to regenerating the external triggers, thus enabling replay of the whole interaction. Replay is not always possible otherwise because of the *feedback* problem (see Section 7.2).

Consider a typical CSCW application; a cooperative shared meeting with multiple users taking part. Providing the new feature of capture and storage of the event messages that are propagated amongst the distributed components can allow:

- *replay of sequences of interaction and collaboration for post-meeting review.*
  A participant, or a person that was unable to take part in the original meeting, can (as allowed by any security policies in use) replay, browse through, and review the whole and/or part of the meeting.
- *locating of the sequences to replay or browse.*
  It is useful for one to be able to search on event type, event attribute, or activity composed of multiple events; with the ability to further qualify each in terms of other events and/or activities that have occurred before or after. This allows users to locate sequences of interest through imprecise qualification based on the knowledge they can recollect regarding when the sequence occurred (as in a memory prosthesis).
- *automated generation of a summary of the meeting.*
  In addition to automated minute taking based on real-time monitoring of activity, it is also possible to extract a limited summary of a meeting from the history of the events that occurred within it. This, however, requires analysis of any video and audio streams used – either through software, by detecting specific images and using speech recognition to pick keywords; or through manual annotation.
- *More elaborate options for coping with mobility and network partitioning.*
  When components become mobile, their ability to receive events is suspended until they reconnect. If the mobile user is involved in a computer-based collaboration, important aspects of the meeting can be lost. For example, when the user's components complete a move to a new workstation, the components can once again communicate with the other users' conference components and network services. At this point, to build their state to a level consistent with that of other users, the components require all the events missed from the period of disconnection. The situation is complicated by the fact that while one user was disconnected, other users may also have been disconnected and thus no user has a consistent view of the application. One approach to address this problem is for each user to retain events from the session so that each has a consistent view of the events generated locally. After re-connection, a user can request that the other users send him/her the events generated by themselves locally.

A similar problem can result if the network fails temporarily, resulting in a conference being partitioned into two or more segments. One way of addressing this without resorting to complex conflict resolution is to show a user what happened whilst he or she was disconnected. This can be carried out by creating a copy of all local objects, initialise them up to the point of partitioning, and to replay within them the collaboration pertaining to the remote conference. Users can discuss what happened within the separated groups during the conference's partitioning, and decide which of the sequences of activity they wish added to the new global session.

While the above points utilise the example of a cooperative meeting, event capture is also applicable in most shared collaboration applications like shared document editing, media spaces [GAV92, BHI93], and virtual-reality multi-user worlds [FBSC93].

## 2.5 Other application areas

This section reviews some further application areas that would benefit from the application of a comprehensive event model and event storage architecture.

### 2.5.1 Visualisation of mobility

Another application enabled by event storage and replay is that of visualisation of user mobility. Several technologies enable people to be tracked and followed as they walk around a building. Examples in deployment include the infra-red-based **Active Badge** [HH94], its ultrasonic equivalent the **Active Bat** [WJH97], and technologies employing image recognition techniques through widely deployed video cameras [Lop99, SKB+98]. The hardware detection technology is then interfaced to monitoring software that exports some bespoke application interface. By interfacing these proprietary technologies to a generic event infrastructure, they can be used together in aggregate for a more thorough view of the physical world.

By retaining a history of movement for some period, several applications become possible. One is, as documented in Section 2.3.1, to enhance the context information within a user's automated diary. Another application is enabling visualisation of mobility for security reasons, or for carrying out safety evaluations of a building. The movements of individuals can be fed into a virtual reality model of the building (see Section 7.6) and reviewed from any three-dimensional viewpoint while, for example, people leave a building during an evacuation drill. This can be used to identify and rectify potentially dangerous bottleneck zones in the layout of the building. Similarly, by replaying people's movements into a knowledge base, it might be possible to deduce the identity of the possible perpetrators of a crime (the classic *who-dunnit* scenario!).

### 2.5.2 The Active Home

An area that is inciting major research and commercial interest is that of the networked home, where ubiquitous computing devices and networked consumer devices are widely deployed. It is proposed here that event notification is the only feasible approach towards internetworking the appliance space. Additionally, event storage can enhance the services that could be made available.

There have been various proposals for enabling the integration of the plethora of devices that one can envisage being available in such an environment (a survey of which is given in [Dut99]), with most exporting a proprietary interface language or adhering to conflicting industry standards. Suitable candidates for networking and digital interfacing in a home are systems like lighting, entertainment (audio and video), security, communications (telephones, video-phones, and door intercoms), climate-control, fridges, ovens, toasters, and even the cat flap! It is unlikely that a tightly coupled communications model can ever be used in this scenario, where devices will be supplied by manufacturers with widely conflicting and competing commercial objectives and backgrounds. The computing capabilities embedded in such devices are also likely to vary considerably in their extent and heterogeneity (see Figure 2.3). With these constraints, a loosely coupled model of communication and interaction is appropriate.

The above, and the fact that most household device programming is reactive in nature ("*do A in response to B*"), makes the event-driven style particularly appropriate.

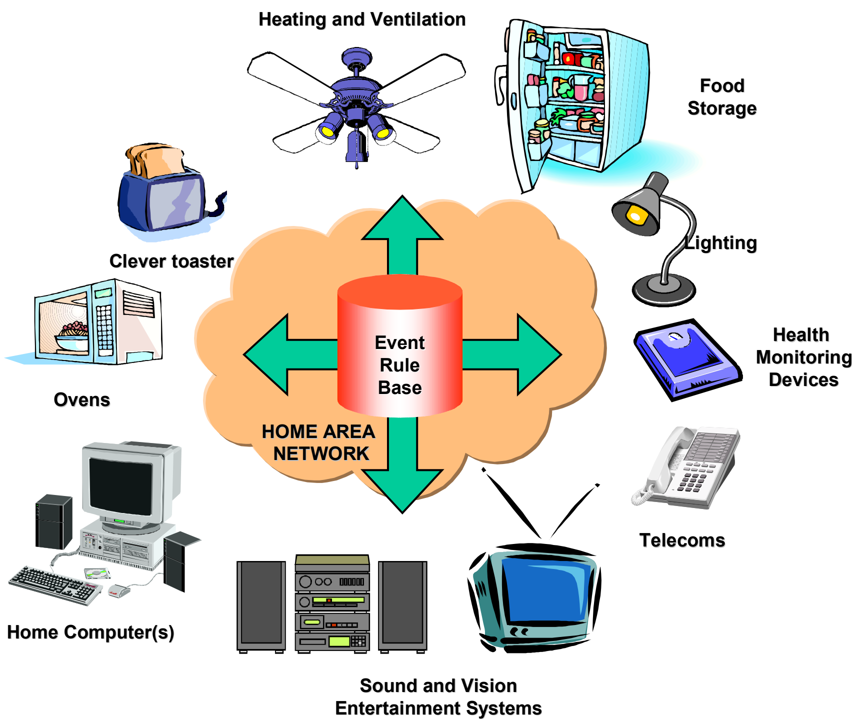

**Figure 2.3**
The Active Home

Lightweight wrapper programs can be written around the proprietary interfaces to make devices *active* (i.e. event sources and/or consumers). Other software modules can then be written to control and monitor the various devices according to some declarative logic. Novel examples of such expressive configurations are *"If I am home, turn on the heating in the kitchen at 6am and activate the hot water boiler"* or *"if I am watching television and someone rings the doorbell, re-route the outside camera image onto the TV"*, as well as *"move the music around the house to follow me as I walk around"*.

Configuring such rules can be complex, and possessing information on behaviour patterns can be very useful. By building a model of behaviour of the residents, as well as monitoring patterns of variation in the physical environment (like temperature), the control software can be made to fine-tune its programming or devise entirely new flexible rules for driving the home's hardware. In short, the *Active Home* can be made to adapt itself to suit its owners' preferences.

### 2.5.3 Graphical user interfaces and usability studies

Although windowing systems and graphical user interfaces (GUIs) are, for the most part, single-address-space applications, event notification and messaging have long been employed within them. The chief concern with the current approaches undertaken by these systems is their reliance on proprietary and largely application-specific centralised event infrastructures. In this area, event storage and trace interpretation has also found some application, however the query interfaces provided are bespoke, based on the application-specific interpretation that is applied. It is therefore not possible to deploy these systems within other environments. The pitfalls encountered by these approaches further motivate this thesis.

Graphical user interfaces of complex applications are usually composed of hundreds of windowing components like scroll-bars, menus and menu entries, push buttons, text-fields, dialog boxes, status bars, drop lists, etc. Constructing a graphical application with multiple layers of windowing containers and elements, and configuring communication between them is complex. For this reason, most GUI development packages model windowing elements as stand-alone objects, and the user interface is then assembled by encapsulating multiple elements within each other as in a structural hierarchy. For example, the menu entries are embedded within a menu option, and the menu options within the menu bar, which in turn is encapsulated within the window header component. Input events pertaining to input devices like the mouse, keyboard or touch pen are passed on upwards along this hierarchy if the components 'register' that they can process, or 'listen to', such events. However, these events often also need to be propagated across the hierarchy, since, for example, moving the mouse over a menu entry could be made to display an explanatory message within a status bar at the bottom of the application window. This can be handled by each windowing element exporting a list of events it can supply, and other components can then designate themselves as listeners for these events. This is analogous to event notification, and a good example of it is **Java 2™**[Sun99c]'s internal event model within the **AWT** [Sun97b] and **SWING** [Sun99a] user-interface construction class packages.

Retaining and storing some or all of these internal events has been applied in a number of contexts. Most notably, in studies of application usability, event traces (usually coupled with captured video) pertaining to users' interaction with an application are employed to appraise its user-interface [BHS93, HHH92]. Keystroke, mouse clicks, mouse trajectories and other low level events are logged and interpreted based on some semantic knowledge of the state of the application and interface. Evaluation parameters are the interface's intuitiveness (how easy it is to figure out how to carry out some operation) and expressive power (how many steps are required to complete a task).

Of particular interest is Microsoft's **Lumiere** [HBH+98] technology that seeks to not only study user interaction but also assist the user in real-time. At the heart of **Lumiere** are Bayesian models that capture the uncertain relationships between the goals and needs of a user and observations about program state, sequences of actions over time, and words in a user's query (based upon when the query was made). **Lumiere** monitors events with an event system that combines atomic actions into higher-level modelled events. The modelled events are variables in a Bayesian model. An event language was developed for building modelled event filters. As a user works, a probability distribution is generated over areas that the user may need assistance with. A probability that the user would not mind being bothered with assistance is also computed. **Lumiere** is deployed in all of Microsoft's **Office 97/2000** products [Mic99b] as the **Office Assistant**. In its production version, an animated avatar is used to suggest context-sensitive tips to users on how to facilitate their work by using features they may not be aware of. The system also attempts to infer questions that users might be about to ask before they have even brought up the help system. In **Office 2000**, the applications monitor users' use of menu entries over time and modify the menu selections available to more closely reflect the feature-set that each individual deems useful. Similarly, the forthcoming **Windows 2000™** operating system [Mic99c] contains a related module that studies several ways in which a user interacts with its windowing environment and dynamically alters its structure to prominently reflect applications and tools that are used frequently.

## 2.6 Conclusion

This chapter has not exhaustively detailed all the possible applications of event notification and storage of event information. Instead, it has introduced the most relevant application categories. Within these broad categories, it has discussed novel and enhanced functionality that can be enabled through the provision of a powerful event storage paradigm, and highlighted related work. Much of the envisaged functionality requires complex application logic that is beyond the scope of this document. Rather, its illustration motivates the requirement for an event storage service that can provide a sufficiently powerful storage and retrieval paradigm that supports tackling the major research issues in these areas.

By examining the core requirements of the functionality described, one can draw up a set of criteria that need to be satisfied by a comprehensive event storage solution. These are:

- *a generic and flexible model for representing event types,*
  Proprietary event-driven solutions sometimes have an event type-system, while at other times rely on unstructured string-based messages. This model is often not extensible, therefore restricting evolution of the application. Invariably, this also implies that the system will be a closed world of related components. A generic event type model, on the other hand, encourages dynamic adaptation of an application, and enables it to be constructed from independent components.
- *a non-intrusive architecture for notification and capture of events at various conceptual locations within an application,*
  Event information can be collected at various locations, for example, at event sources, at centralised application-driven locations, and at gateways between domains. Event storage also needs to be employed within the event notification mechanism as well as outside it, interacting directly with applications. Issues like lack of global time, network latencies and scalability need to be addressed.
- *an event storage architecture,*
  An event repository must be flexible in its design, so that it can be customised for environments with constraints on memory, storage and processor availability. It must also embed a storage engine that can process and keep up with streams of incoming events, while at the same time provide facilities that enable a temporal retrieval interface to be deployed over it.
- *a core subset of temporal retrieval and replay functionality that can be built-upon by applications to enable higher-level application-specific interpretation of event histories.*
  In order to enable interpretation and analysis of event histories, a retrieval interface is required that goes beyond traditional database query languages and supports primitives that reflect the temporal sequencing of events.

These requirements and the issues they raise are addressed in depth in Chapters 4, 5 and 6.

Chapter 3

# Related Event Middleware

This chapter reviews related general-purpose event-driven and messaging infrastructures.

It starts by briefly introducing distributed middleware and the important role it plays in building distributed systems. It then focuses on asynchronous messaging technologies that implement event notification. While Chapter 2 dealt with event notification at application level by describing its usage in various domains, this chapter looks more closely at specific messaging and event-driven systems, and their infrastructures.

Section 3.1 lists the technologies available in distributed application middleware and describes coupling models of integration. Section 3.2 then shifts the focus onto event notification by describing variations on the event-based messaging model.

Two primary categories of event transports are then distinguished; event-driven systems are reviewed in Section 3.3 while message-oriented middleware products are treated in Section 3.4. Although the functionality of either overlaps considerably, their distinct origins and different (although converging) philosophies merit separate discussion. The most representative solutions and products in both fields are reviewed with a view to their application, or lack, of event storage and retrieval.

## 3.1 Distributed application middleware

In general, building any distributed system implies satisfying a number of basic requirements; application processes are distributed across several machines, those processes need to be located, and they need to communicate amongst themselves. In addition one needs to maintain security of the data, work with heterogeneous platforms involving different networks, operating systems and data formats, and cope with limiting constraints like unreliable, low bandwidth or high latency network connections. Addressing all these issues within each application is a major task, and therefore it is practical to resort to general-purpose middleware that tackles the problems transparently and abstracts away all the nitty-gritty details of distribution.

In this discussion, therefore, the term *distributed middleware* is taken to denote software components, running between the application and operating system layers, that address the above issues and enable communication complexity to be abstracted away from the application writer. This is usually carried out by providing the application developer with an Application-Programming-Interface (API) that abstracts away most aspects of distribution.

## 3.1.1 The categories of middleware

Distributed middleware can be grouped into three loosely defined categories:

- *Client/Server*
  This involves connection-based communications using a procedure-oriented invocation model. Examples of this category are remote database access middleware, remote stored procedures and remote procedure calls (RPC). In this well-established method of distributing application processing, an application uses RPC's to execute remote procedures that are located in another program, where the latter can be located on a different computer and/or platform. Client/Server uses the call procedure construct from structured procedural coding techniques. The client calls a procedure to perform an operation, which is then carried out by the server on behalf of the client. Execution at the server is thus controlled by the client. RPC communications are inherently synchronous, with control being passed from the local procedure to the remote one, and local execution being blocked until a result (and control) is returned. Asynchronous communications can be supported but require explicit support from the application through multi-threading.

  RPC systems emphasise strong data-typing of the result data transferred. This is due to their frequent integration with a programming language environment that is common to both client and server. The server's interface is mirrored in a *stub* or *skeleton*, that is then linked with the client code to enforce type matching at compile time. This is illustrated in Figure 3.1.

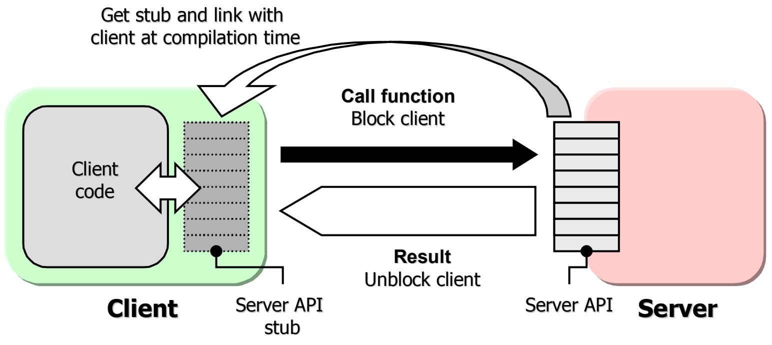

**Figure 3.1**
Client/Server – Remote Procedure Call

There are a large number of proprietary RPC systems, with the most widely used being DCE's RPC service [OG97], **ONC** [Sun98], and **Java™ RMI** [Sun97a] (whose features make it overlap onto the next category below).

- *Distributed object frameworks*
These provide connection-based communications using an object-oriented invocation model to distributed computing, which increases the flexibility and reusability of application systems. In the same way as client/server, the client controls the operation of the server. The user performs operations through a local object on an object created by the distributed object system, by invoking its methods. In general, this interaction is also synchronous. The distributed components, here being objects, usually interact through an object request broker, or ORB. The ORB handles the requests that an object makes of another object, and provides the mechanism for locating and interacting with objects across the network. The structure of the OMG's **CORBA ORB** [OMG99] is illustrated in Figure 3.2.

  Distributed object frameworks enforce strong data-typing but are designed to integrate heterogeneous components with different environment type systems. In order to address this, they provide a standard type system that language-specific type systems map to. An interface definition language, like the Object Management Group (OMG)'s **IDL** [OMG99], is used to describe the object interfaces, from which programming-language dependent stubs can be compiled. These then allow compile-time type checking.

  The main distributed object technologies in widespread use at present are the OMG's **CORBA** standard, and Microsoft's **DCOM/COM+** [EE99].

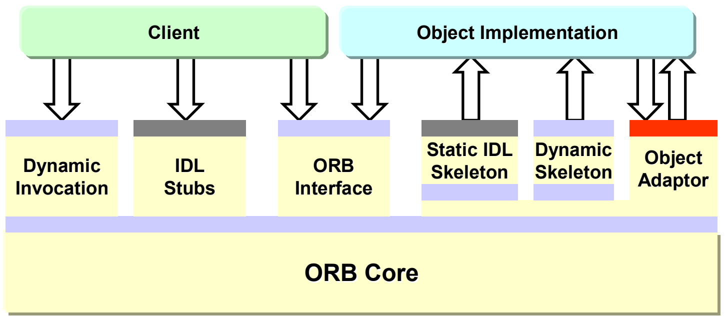

**Figure 3.2**
The structure of a **CORBA Object Request Broker**

- *Event-driven systems and Message-Oriented Middleware (MOM)*
Message-oriented middleware usually involves connectionless communications using a message transport, sometimes called an *event bus architecture* or *message pipeline*, to send event messages between applications. An asynchronous peer-to-peer invocation model is employed, where an application sends a message by passing it to the local middleware at its end. This step is functionally separate and decoupled from the act of transferring the message to its destination, where the receiving end of the messaging middleware delivers it to the receiving application. No acknowledgement of delivery is sent to the sending application

unless it explicitly asks for one. Building on the basic communications channel thus provided, important value-added services include *message-queuing middleware*, where guarantees can be provided on message transfer even if the applications cannot communicate at all times. Another important service is that provided by *message brokers*, which amongst other things, provide for *publish/subscribe* services, message transformation, and limited composition. Several MOM systems do not have a type system for structuring message data. There are several industrial products providing MOM, and Section 3.4 illustrates the functionality of some of the major products in the market.

Pure event-driven systems have evolved separately from message-oriented middleware; whereas MOM originated from the need to reliably connect application programs to a centralised database server, the concepts behind event notification systems derive from internal system tasks like graphical windowing and active database triggers. In this area, the emphasis is on a *source* of event messages advertising what events it has available, and *clients* then registering their interest in being notified on when those events, with some qualifying constrains, occur. The constraints can reflect content, frequency, and composition. In its most lightweight form, this corresponds to the *publish/subscribe* service provided by most MOM vendors. However, several research systems go far beyond this by supporting type systems, comprehensive registration, event brokering, federation, and composite event sequence specification. Section 3.3 describes some examples of this functionality in more detail.

Although the above categories might appear exclusive, in practice several middleware solutions in use today provide functionality overlapping across all three areas. No technology is more exemplary of this that the ever-evolving Object Management Group's **CORBA** specification, which now provides services to tie it into RPC-based systems, messaging, and supports event propagation through the **CORBA Event Service** [OMG97] and the **CORBA Notification Service** [OMG98a] specifications.

### 3.1.2 Distributed models of coupling

Client/Server and distributed object frameworks are based on a *tight coupling* between the object that requests a service (the client) and the object that satisfies such requests (the server). Before invoking a service, the client has to know the existence of a server capable of satisfying its request and has to obtain a reference to the server. These client and server need to be:

- aware of each other's API. Changes to any one side need to be applied or reflected within the other application,
- online at the same time,
- able to communicate *synchronously* over a network.

These constraints are difficult to address in dynamic environments, where the clients and servers making up a distributed system have different lifetimes and may be under different technical or organisational management. In many situations, a de-coupled, or *loosely coupled*, communication model is preferable. Under such a model, communicating distributed modules (not necessarily identified as clients and servers):

- need have little or no knowledge about one another,
- can be modified independently of each other without requiring modification or recompilation of one another,
- need not run at the same time,
- communicate *asynchronously*, in that they can proceed with their computation independent of each other's state of execution and need not block awaiting replies to messages.

Message-oriented middleware and event-driven systems meet these criteria by decoupling the delivery of data from the distributed components. This enables dynamic system configuration. The remainder of this chapter will now focus on these systems and illustrate them in further detail.

## 3.2 Event-based messaging models

There are several variations on the event-based messaging model (as defined in Chapters 1 and 2), most relating to the level of distribution of the services making up the event-based style and their location. There are three notional entities in the event-based model: (1) *a source* of messages or events, (2) *a consumer* (or client) of events, and (3) *a message transfer service* that delivers messages from the former to the latter. The services of event-registration (or subscription), filtering, action injection, fault-tolerance/error handling, and message transformation that may be available can be positioned at various locations at or in between these three entities. Different systems distribute or centralise these services differently. For example, filtering can be carried out either at the source, at the message bus through a notification server or a message broker, or at the client. There are advantages and disadvantages to each approach.

From amongst these distinctive approaches, one can identify two opposing styles:

- *Direct (Client/Source)* – the event clients and event sources are aware of each other during communications, that is, a source sends a message directly to a client it knows about, and clients carry out registrations directly with the source of information. This does not imply a static system, as the components can discover information about each other dynamically at runtime. This is the pure distributed event notification model where there is no centralised messaging entity (except maybe for a directory service or other specialised components). The core messaging technology is therefore available at each component, and therefore, the functional focus of the system is in its components.
- *Indirect (Client/Channel/Source)* – in this model all functionality is contained in the delivery entity, the *event/message channel* (also called a *message bus* or a *message transport*). This can be distributed, centralised, or consist of a mixture of

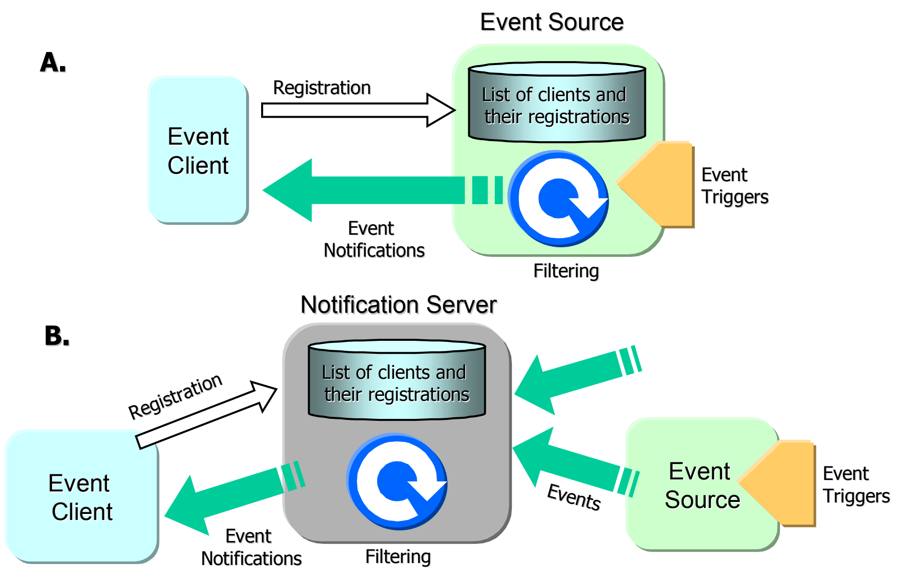

**Figure 3.3**
A. Direct event model
B. Indirect event model

distributed and centralised modules. Therefore, the source passes on its messages to the channel, and clients register with the channel. Clients and Sources know only about the message channel and do not know about each other. The message channel is therefore a complete go-between.

These two approaches to event notification are shown in Figure 3.3. In practice, many event systems have elements of both styles, and the same functionality can be provided by both. While in the indirect model, messaging functionality is embodied in the transport, and duplication of functionality can be removed through centralisation, the direct approach requires that all functionality be replicated at each component. Although this may require that platforms at the component level be more powerful, it makes integration of components under different management more straightforward and renders the system immune to failures of crucial modules.

Another important qualification is how general purpose an event notification system is. Most of the event notification systems deployed in the areas outlined in Sections 2.2 - 2.5 are designs tailored for the application domain they are deployed in. An indication of a design's generic nature is whether it applies any semantic meaning to the event messages it delivers. A general-purpose event solution should not make any assumptions as to the meaning of the event data it is delivering and should not attempt any non-application-driven interpretation of it.

Early event systems did not provide type-support, and messages consisted of strings or packets of unstructured bytes. Since then varying degrees of type system support have been provided. Some transports only allow message typing, in that an event message can be of a defined type. At the other end are comprehensive type systems that allow structured typed events with typed parameters. In the latter, event schema can be propagated amongst components statically through stub linking at compilation time, as in **CEA** [BBHM95], or dynamically at run-time, as in HERALD (see Chapter 4).

## 3.3 Event-driven systems

There are several systems that qualify as general purpose event-based communication mechanisms, with some having evolved into such from more bespoke origins in specific application areas (see Chapter 2).

Several early software integration systems, such as **FIELD** [Rei90] and **SoftBench** [Ger90], provide message routing services to deliver messages and enable modules to react to events generated elsewhere. Event-based general-purpose communication mechanisms for loosely coupled integration are suggested in [GN91, SN92, OPSS93, MDL93].

The concepts of event notification have found application within programming environments. An event-based language for parallel programming called **EBL** is described by Reuveni [Reu80]. In this language, events are the only control mechanism and cause the activation of event handlers. Some languages have event announcement primitives built in to facilitate intra-program interaction, e.g. **Smalltalk** [Sha89]. **ECO** [SCT95] defines language extensions for **C++** and **Java™** to enable event-driven method invocation in objects.

Similar in concept is the idea of *tuple spaces*. Tuple space systems define a shared space, like a whiteboard, where *tuples* (a list of typed data fields) of information are placed. Multiple concurrent client processes then have available a small set of operations through which they can interact with the tuple space, and read and write data in it. A sender places a tuple in the tuple space, and receivers can inspect or remove tuples from this space by specifying a template tuple. Reception occurs when a match for the template tuple is found. **Linda** [Gel85] first illustrated the concept of a tuple space, and since then, several other systems [MW88, DFWB98, WMLF98] have been proposed that enhance the basic functionality by providing services like persistence and distribution.

Event notification concepts have a long history in the AI community, such as *actors* [HI91] and *blackboard systems* [JDB89]. Some rule-based systems, such as those found in certain process-centred environments like **Darwin** [MR90], **Marvel/Oz** [BK95], **Oikos** [MA94], and **Adele** [BEM94], are based in part on event-based substrates, in that updates to data may trigger particular actions. Generally, these systems have a broader focus than application integration, encompassing configuration management, software process and other domains.

This section introduces some general-purpose distributed event systems and illustrates their main features. Where applicable it describes their use and support of event storage.

## The Cambridge Event Architecture (CEA)

The Cambridge Event Architecture (**CEA**) [BBHM95] is a *direct* notification model (as defined in Section 3.2) and was one of the first architectures to embrace the *publish-register-notify* programming paradigm. **CEA** has been used in building of large scalable distributed systems in various fields, amongst which were multimedia cooperative applications, telecom monitoring systems [Ma97], network management and location-oriented applications [BHB96, BSHB98].

Just as **CORBA** objects are defined using the OMG's **Interface Definition Language** (**IDL**) [OMG99], which allows other objects that want to interface with them to encapsulate pre-written stubs at compilation time, active objects in **CEA** use an extended version of **IDL** to declare and publish the events they will notify to clients if asked. Event stubs are automatically generated and linked in with client code. Such objects have a *register* method in their interface and interested parties may register interest in any event class, specifying parameters or wildcards in their *registration template*. Event occurrences are created as objects of specified type. Events may be named and parameterised, where the parameters can be of any **IDL** supported type. When an event occurs, the service matches it against a stored template associated with each registration; subject to access restrictions, each client whose template is matched will be notified of the event that has occurred. Figure 3.4

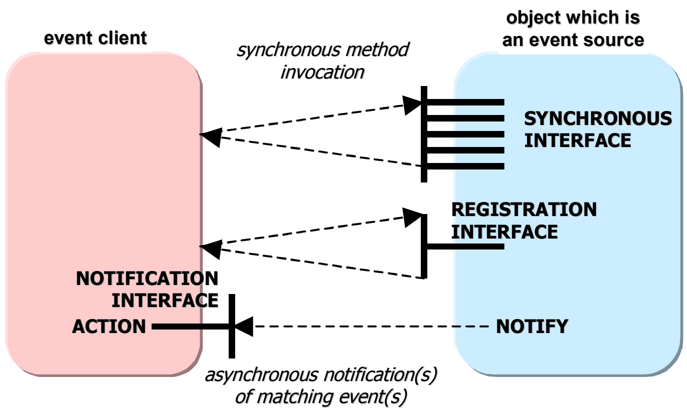
event client
synchronous method invocation
object which is an event source
SYNCHRONOUS INTERFACE
REGISTRATION INTERFACE
NOTIFICATION INTERFACE
ACTION
NOTIFY
asynchronous notification(s) of matching event(s)

**Figure 3.4**
The publish, register, notify event architecture of **CEA**

shows the approach.

In addition to *direct*, source to client, notification of events, intermediate services known as *event mediators* can be defined. An event mediation service might be set up to notify any number of clients, and might register interest with any number of event sources. One use for a mediator is to remove the filtering function from a primitive event source by providing an indirection between it and its potential clients. Any source that cannot afford the overhead of template matching can notify all its detected events to such a mediator. A mediator can be used to prevent a mobile user from missing events of interest while disconnected from the networked systems for periods of time. It registers interest with the required event sources on behalf of the mobile client and buffers the events notified to it by these sources. It also registers interest in the location of the mobile client, and notification of an *attach* event (detecting the mobile user) triggers the delivery of the accumulated events to the user at the new location. **CEA** therefore employs source-side filtering in order to reduce the volume of events sent over a network and in order to ensure that clients need only receive information that is relevant to them. This also reduces performance requirements at the clients.

**CEA** also introduced the notion of composite events being generated from temporal combination of primitive events. Complex scenarios can then be defined and specified, and composite event services can be built that will, on their clients' behalf, register interest with appropriate event sources and notify clients when a composite event is detected. A *composite event server* is an example of an event mediator that can carry out filtering across events of different types from different sources.

The **CEA** architecture has been designed to inter-work with a comprehensive role-based access control scheme called **OASIS** [Hay96].

In [BBMS98], the **CEA** event model is enhanced to support hierarchical event type specification and inheritance, and the concept of an *event federator* is introduced. This module can detect composite event sequences and inject actions into application components in response. Event brokers also retain information pertaining to the event sources and can act as gateways between different event systems, providing event transformation and mapping services. The HERALD event transport (see Chapter 4) builds on this extended framework.

Due to its reliance on **IDL** and static stub linking, **CEA** cannot be termed a loosely-coupled infrastructure as defined in Section 3.1.2. This is because event client objects that want to interact with event sources need to know about each other's interface methods at compilation time. Its primary contribution is in demonstrating how asynchronous operation and comprehensive event services can be added to inherently synchronous industry standard platforms. **CEA** implementations have been demonstrated on top of RPC-based mechanisms like **MSRPC3** and **Java™ RMI** [Sun97a]), as well as on top of **CORBA**; the **COBEA** implementation [MB98].

## The Events-Constraints-Objects (ECO) model

In the **ECO** (Events-Constraints-Objects) [SCT95] programming model from Trinity College, Dublin, the basic abstractions are *objects*, *classes*, *events* and *constraints*. **ECO** is a programming model rather than a middleware service, as it defines event extensions to object-oriented programming languages like **C++** and **Java**. Its treatment of events as triggers for method invocation is reminiscent of **Smalltalk** [Sha89].

In **ECO**, objects are instances of a class, and have instance variables and methods that operate on those variables. A class specifies the interface to its instances together with the events and constraints used by the instances. A property of the class is therefore its capability to consume and to generate specified events. Objects communicate by announcing events and by processing those events that have been announced. A method can be bound to one or more events and several methods of an object can be bound to the same event. A bound method therefore behaves as an event handler in that it is invoked automatically when the corresponding event is announced. The programming language runtime system is responsible for gathering the events generated and for notifying events to the interested objects. This effectively decouples objects, in that they do not have to be aware of each other and can execute asynchronously with respect to each other.

A constraint specifies a condition that controls the propagation of events. There are different kinds of constraints, categorised by the data that they can access, by their evaluation points, and by the actions that they are allowed to perform. Notify constraints are set by destination objects at subscribe time, and place conditions on notification. Such conditions are priority and the number of recipients. Pre- and Post- constraints are attached to the event/method bindings and can be employed to force synchronisation in an otherwise fully concurrent programming environment.

## Other event systems

**FIELD** [Rei90] was probably the first event-based integration system. **FIELD** has a client-server architecture. Client programs, called *tools*, broadcast messages anonymously by sending them to a central message server, **Msg**. Tools specify the classes of messages that they want to receive by registering message patterns with the server. **Msg** then delivers to each tool only the messages that match the tool's message patterns. **FIELD** also provides an optional tool, called the **Policy Tool**, which can intercept messages and perform user-specified actions on receipt of those messages (reroute messages, replace them with other messages, etc). These user-specified actions, called *policies*, are external to tools, so no tool modification is necessary to enforce policies.

**Polylith** [Pur94] employs a *software bus architecture*. Individual programs, also called tools, connect their input and output ports to an abstract bus, and send and receive messages on named bus channels. A module interconnection language (**MIL**) is used to encapsulate external programs as tools and then bind the output ports of tools to the input ports of other tools. Messages may be of simple, structured, or pointer types.

In **Elvin** [FMKAPS99], notifications are sets of named and typed data elements. A subscription is a declarative Boolean expression over the contents of event notifications. By issuing a subscription, a component can declare its interest in a number of notifications characterised by some common property.

A similar mechanism is provided in **JEDI** [CDF98]. In **JEDI**, a notification is defined by a name and by a number of parameters as in **CEA**. Event receivers subscribe for event patterns, which are expressions over the name and parameters of a notification. This is more expressive than the template matching mechanism of **CEA** in that subscriptions can also apply wildcard match on the event type itself as well as on the *number* of parameters.

**Yeast** [KR95] is a client-server system in which distributed clients submit *event-action* specifications with a centralised server, which performs event detection and specification management. Each specification submitted by a client defines a pattern of events that is of interest to the client's application plus an action that is to be executed in response to an occurrence of the event pattern. Therefore, this is not an event-based infrastructure per se, since its event service triggers actions rather then notifications. By default, any output produced by the commands of the action is sent by electronic mail to the user who submitted the specification. Most of these specifications and events relate to platform and operating system properties like filing, disk storage, memory usage, processor load, user activity etc. **READY** [GKP98] elaborates on **Yeast**'s specification language to allow compound matching and aggregation (similarly to **CEA**).

## 3.3.1 Event standardisation efforts

There are a number of mainstream event-related standardisation efforts. The main ones are the OMG's **CORBA Event Service** [OMG97] and **Notification Service** [OMG98a] specifications, the TINA consortium's **Notification Service** [TIN96], and the IETF's **Internet-Scale Event Notification Service** effort that is still at the drafting stage.

### CORBA Event Service

The **CORBA Event Service** specification defines an indirect channel-based event transport for deployment within distributed object frameworks. It defines an **EventChannel** interface that decouples event *suppliers* and *consumers*. Suppliers cause events to be generated and placed onto the channel, and consumers obtain events from the channel. Channels are typed, in that a channel only has the means to signal events of its own type to its consumers and will ignore all other types of events. In the **Event Service**, as in **ECO**, and in contrast to **CEA**, events are not objects or entities onto themselves. They are merely triggers that cause methods to be executed, albeit untyped parameters can be passed into those methods. The specification allows for different styles of interaction with an event channel, specifically *push* and *pull*.

*Push interaction* is supplier-driven. In this style of interaction a consumer object must have its interface conforming to the **PushConsumer** interface. After it registers

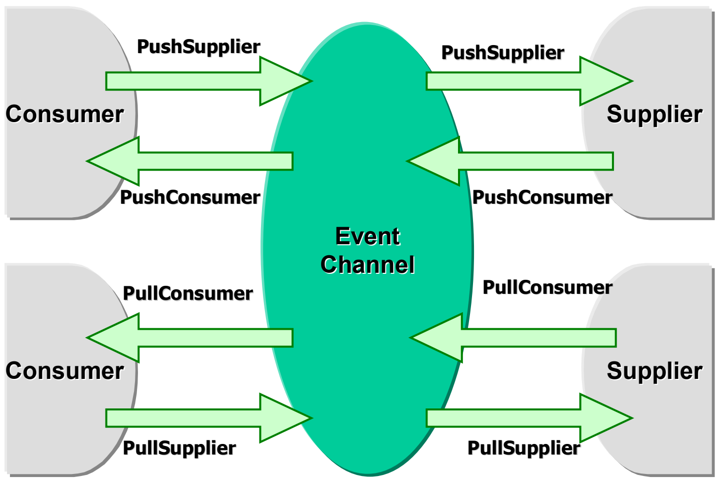

**Figure 3.5**
Event propagation styles in CORBA

interest in an event type with the channel, whenever an event of that type is generated at a supplier, the consumer's **push** method is invoked with the event's data as input parameter.

On the other hand, *pull interaction* is a polling consumer-driven style. In this style the supplier buffers events of the type that a consumer has registered interest in, and it is up to the consumer to invoke a **pull** method each time it is ready to receive an event. This blocks the client until such an event is available, upon which the method is executed with some or no event data. An alternative non-blocking interface that returns a *boolean* value reflecting whether an event was available or not can also be employed. Event buffering is initiated at the supplier once the pull-style client-channel-supplier connection is made.

These two styles of event propagation are illustrated in Figure 3.5. A channel can support multiple suppliers and consumers concurrently.

The **CORBA Event Service** specification does not directly support events having structure and typed parameters, and therefore restricts the extent to which a consumer can define what events it wants to receive. Its decoupling of supplier and consumer identity is also not always desirable.

## CORBA Notification Service

The **Notification Service** specification is currently at the draft stage, but is close enough to being finalised to warrant discussion. This specification is important as it addresses the shortcomings of the **Event Service** specification and extends it to provide features from research-based event systems. Its main developments in this regard are in supporting structured events, content-based filtering, and a quality-of-service (QoS) interface.

In the **Notification Service,** an event is similar to an object in that it is structured into a number of compulsory and user-specified parameters. It has a type defined as a **domain/type/name** triple, and within a secondary header defines a number of optional constructs that define its priority, reliability and timestamp. The user-specified body is divided in two sections, each of which can contain any number of fields. The first lists filterable fields that may be used by a consumer to specify filtering, while the second contains additional message parameter data as desired by the user.

Channels support the filtering interface, which allows a filter object to be attached to them. Such an object, usually located before a consumer, only forwards through it events that match the filtering expressions it is configured with. Mapping filters may also be used that take action on events according to their optional headers.

QoS constrains may be imposed on a per-channel, per-proxy and per-event basis. The specification demands that the following default set are recognised (although not implemented): **Connection Reliability**, **Priority**, **Event Reliability**, **Maximum Batch Size**, **Ordering Policy**, **Discard Policy**, **Expiry Time** and **Pacing Interval**.

Finally, the specification goes some way to support dynamic evolution of events and dynamic adaptation of consumers by providing for an **Event Type Repository**. This facility lists the names of event types with their associated structure.

### 3.3.2 Storage in event systems

To the best knowledge of the author, none of the above general-purpose event systems or service specifications have defined an event storage service. The need was perceived in the context of the **CEA** research project at Cambridge, and this dissertation is a result of that research. However, an analysis of the underlying concepts behind most of the above event-driven systems reveals scope for not only the provision of an event storage and retrieval service within event-based systems, but also for integration of such a service within the core infrastructure of the systems. This is highlighted in the discussion of HERALD (see Chapter 4).

## 3.4 Message-oriented-middleware

This section describes Message-Oriented-Middleware (MOM), the mainstream commercial equivalents of event-based systems. Early MOM products originated from the need for remote applications to communicate asynchronously with centralised database systems, but rapidly evolved into middleware for enabling systems integration (particularly in terms of legacy applications). The term 'messaging' is popularly used to refer to electronic mail systems, and this comparison is appropriate since messaging middleware is analogous to a general-purpose high-speed e-mail system with guaranteed delivery.

### 3.4.1 Core features of MOM

MOM provides an asynchronous communication channel (or *message bus*) between applications or components. There is no explicit client/server relationship at the middleware level, as the communicating applications are viewed as equals. One application sends a message to another. Its primary acknowledgement of delivery is that it has submitted its message to the messaging middleware at its side. The messaging transport then guarantees that at some time, in some way and through some route, the message will be delivered to the intended recipient. On top of this, some products/systems do provide temporal guarantees on delivery, and the ability to request acknowledgement of receipt from the receiving application. This structure is shown in Figure 3.6.

*Message-Queuing Middleware* (MQM) is a type of MOM that combines a message transport and a queuing service. Although, conceptually, message queuing is a service provided on top of the core messaging transport, it is required for guaranteed delivery and has become an intrinsic feature of messaging. Message queuing is provided by all major MOM products. Messages are pushed into the local transmission queue for delivery, delivered between queues asynchronously, and placed into the receiving queue at the receiving end. The receiving application can then retrieve them when it can. Transactions can be applied to the act of placing and retrieving a message into a queue, transmission between queues, or over all three

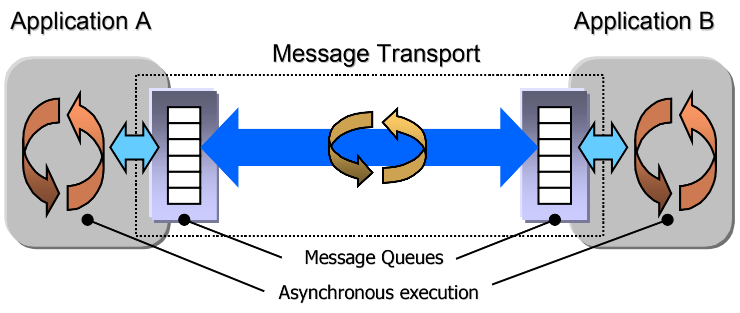

**Figure 3.6**
Decoupling of message propagation in MOM

stages. Queues can be persistent on disk storage, and can be reinstated after a system failure or restart. Other key functionality includes load balancing, multi-casting, guaranteed delivery, and several levels of fault tolerance. The various MOM products available provide some subset of this functionality, and most add several high-level value added services like *message brokering* and *publish/subscribe.*

A message usually consists of a structured collection of fields making up an object or data structure, mapped to a human- and/or machine-readable string. The nature of the mapping employed and extent of the typing supported varies according to the messaging system, and can be language or platform dependent. At present, there is a drive to employ **XML** [BPS99] tagging as a standard representative format, due to **XML** data being self-describing. Although messaging is particularly suited for application integration over heterogeneous platforms and environments, it can be applied over homogenous domains, and as a communications pipeline for other middleware technologies. For example, two applications built using the same programming languages (like **Java**™ or **C++**) can transmit *flattened* (or *serialised*) objects over MOM. Similarly, the *marshalled* parameters of a **CORBA** remote invocation may be sent over a messaging transport. In fact, the **CORBA Messaging** specification [OMG98b] lays down a standard framework for carrying this out.

Products from different vendors have different APIs and do not easily interoperate with one another. Each vendor's products employ proprietary location (directory and naming) and security services. Sometimes there are, however, gateways that make it possible for competing products to communicate with one another, particularly with regards to the major market players.

## 3.4.2 Publish/Subscribe

At its core, message-oriented-middleware is concerned with delivering messages from one application to another. Through a *publish/subscribe* service, the system can then enable message delivery to be tailored for recipients that it knows have expressed some interest in a category of messages. Publish/subscribe is therefore analogous to *publish-register-notify* as discussed in Section 3.3. Information is published, or advertised as being available, and clients can then register interest in being notified of when messages matching their subscriptions occur. The propagation of messages is

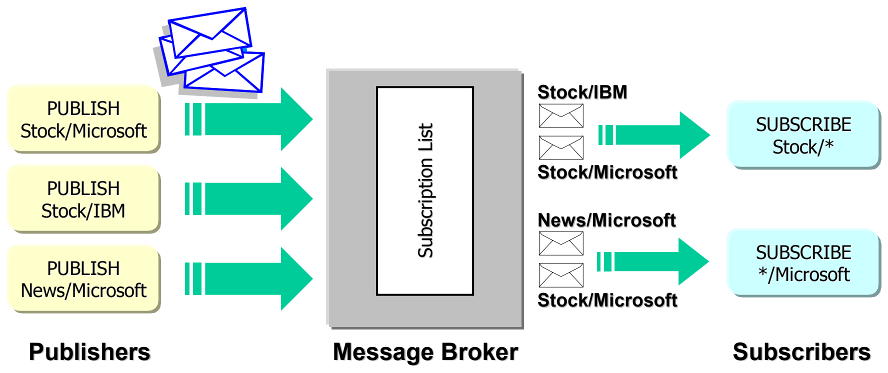

**Figure 3.7**
**MQSeries™**'s Message Broker.

therefore dependent on subscriptions that may be defined and revoked by client applications at any time. The implication of this is substantial, as client applications can specify information they want to be told about, rather than receive everything that a sending application transmits.

The subscriptions that define what information an application is interested in vary in form according to the particular product or solution, but are usually topic and/or content-based. Messages can be tagged as belonging to some category (which can be part of a hierarchical information structure) through textual keywords. A subscription then defines some filtering criteria to select from the messages available. Figure 3.7 illustrates how subscriptions are defined in **MQSeries™**.

Some message-oriented middleware solutions provide publish/subscribe services, usually through the provision of a *message broker*. The broker conceptually acts as a proxy around the sending application, by accepting its messages, and placing them within some structural category based on their topic/content. Client applications can carry out subscriptions with the broker for specific information, which then dispatches the messages, using the underlying MOM transport, to those applications when the relevant information becomes available.

## 3.4.3 Representative solutions

The market for MOM is substantial and there are a large number of products in wide deployment. Leading products include IBM's **MQSeries™** suite of middleware solutions [IBM99], Microsoft **MSMQ** [Mic99a], Talarian **SmartSockets™** [Tal98], 4Tier Systems's **OpenMOM™** [4Ti99], Oracle's **Oracle8i AQ™** [Ora99] and Tibco's **TIB™** product family [Tib99]. There are no pervasive standards for MOM, although the **CORBA Messaging** [OMG98b] specification is attempting to standardise the integration of MOM with distributed object technology. Some products embrace multiple middleware paradigms and enable wider enterprise level integration. For example, Inprise's **Entera/QX™** [Inp99] provides a platform for integrating MOM and Client/Server, while **TIB™** provides both real-time messaging and message queuing for guaranteed delivery as well as a tightly integrated ORB, which is **CORBA 2.0** [OMG99] compliant.

While the above discussion has highlighted the main features common to most products, it is informative to review the feature-set of some of the main solutions. **MQSeries™**, **MSMQ**, **Oracle8i AQ™** and **SmartSockets™** will now be outlined in further detail. The **MQSeries™** suite dominates the marketplace, and while acting as the benchmark product range, its feature-set is largely representative of the most comprehensive solutions available. **MSMQ** is the main competitor to **MQSeries™** on the important Microsoft **Windows™** platform. **Oracle8i AQ™** has been picked out because of its distinctive integration with a database management system, while the **SmartSockets™** product is particularly relevant to this discussion as it actually attempts to provide an event storage service.

## IBM MQSeries™ and Integrator

IBM's **MQSeries™** suite of middleware software represents some of the earliest available commercial messaging solutions, and today still dominates the market. **MQSeries™** is the MOM transport and, partly due to its availability on 40+ platforms as well as its technical feature-set, controls in excess of 50% percent of the market.

**MQSeries™** applications (local or remote) communicate by putting messages on queues and by taking messages out from queues by using the *Message Queue Interface* (MQI) and **MQSeries™**'s API. A message from a sending application is placed on a queue, where it then waits for a signal that the receiving queue is ready to accept it. Maintaining the messages queues, the relationships between programs and queues, handling network restarts and moving messages around the network, is the responsibility of **MQSeries™**. **MQSeries™** provides various transaction guarantees, which can be controlled by its own built-in transaction processing (TP) monitor or by an external X/Open compliant TP monitor.

A queue can be predefined or created dynamically. A *local queue* belongs to the same queue manager as the application that is connected to it. A *remote queue* is owned by a different queue manager. A *transmission queue* is a special queue used by the queue manager and transmission programs to temporarily store messages destined for a remote queue manager. An *alias queue* provides an alternate name for a queue. A *model queue* is not used directly as a queue, but is used as an example when assigning characteristics to a dynamic queue. Each message queue belongs to a queue manager. The *Message Queue Manager* is the most important and central object in the **MQSeries™** environment. Queue managers are the system providers of message queuing facilities used by applications. Additionally, queue managers process system commands and manage all message queuing related objects. There must be at least one queue manager on a system.

Applications local to one queue manager can put messages on remote queues owned by another queue manager through distributed queue management. Current versions of **MQSeries™** simplify distributed queue management by allowing applications to read only local queues, while allowing them to write to both local and remote queues. Queue managers on different computing platforms communicate with each other through programs, called message channels. They are made up of two *Message Channel Agents* (MCAs or movers), consisting of a sender and a receiver, and a communication link. MCAs communicate with each other using the **MQSeries™** *Message Channel Protocol.*

**MQSeries™** provides assured message delivery by moving messages to remote queue managers in groups. Messages that are defined as persistent are never deleted at the transmitting queue manager until their confirmation of receipt at the destination queue manager. Messages in **MQSeries™** can have a priority field that can override their submission order with regards to their delivery.

IBM's **Message Broker** builds on top of **MQSeries™** ' reliable asynchronous messaging graph to provide publish/subscribe. The broker is an integral part of a queue manager, and keeps track of both *published* information and *subscriptions* to it. The broker can send information to other brokers when required. Subscriptions are topic-based and are defined using character strings. Structure can be implied by the use of delimiters (like '/'). For example "Stock/*" matches published 'stock prices' for 'all' companies, while "News/IBM" would refer to published 'news' relating to 'IBM'. Figure 3.7 illustrated this functionality.

**MQSeries™ Integrator** employs **MQSeries™** as the messaging transport and adds; **Formatter**, which provides message parsing and reformatting capabilities, **Rules**, which provides content-based decision making capabilities, and the **MQSeries™ Integrator** daemon, which combines the messaging, **Rules** and **Formatter** components to process messages. This enables limited monitoring for composite events.

**MQSeries™** does not provide a dedicated event storage service other than making events persistent for some time, until they are acknowledged as having been delivered. There is no distinctive query interface to these persistent queues.

## Microsoft MSMQ

**Windows™ NT 4.0 Server** and **Windows™ 2000** contain **Microsoft Message Queue Server** (**MSMQ**) as a built-in service. The aim of this MOM product is to become the middleware of choice for integrating applications running on the widely-used **Windows™** platform. While its MOM feature-set is fairly mainstream, its attraction derives from its ease of deployment and administration. It benefits from close integration with other operating system services like transaction support, directory service, security and clustering. Obviously, this tight integration with the comprehensive services available in the **NT** domain make it a strong contender in the **Windows™** platform, but also prevent it from being of major relevance in other platforms or in heterogeneous application integration.

## Oracle8i AQ™

Oracle's **Oracle8i Advanced Queuing™** (**AQ**) is interesting because of its distinctive architecture. Given that most of the architectural and storage requirements of persistent message queues are easily available in a relational database, and given Oracle's in-house expertise in relational database management systems,

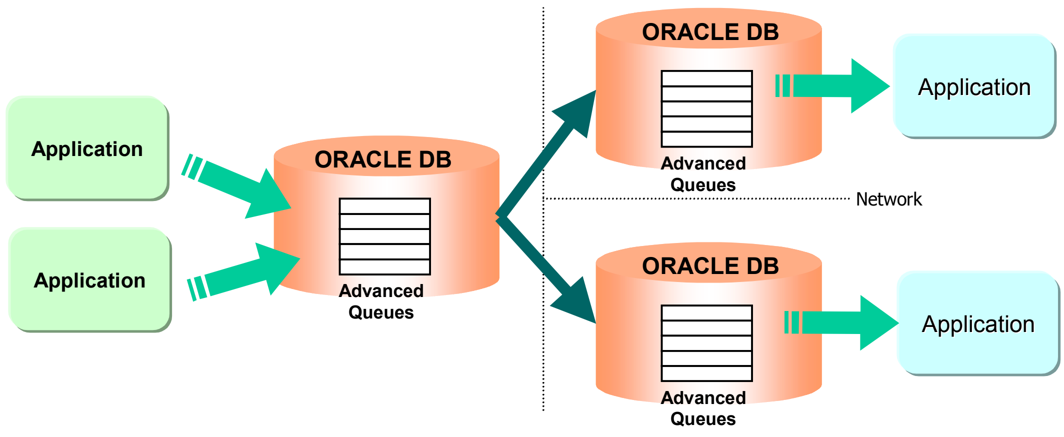

**Figure 3.8**
**Advanced Queueing™** transparently propagates messages over the network in between **Oracle™** instances.

**Advanced Queuing** is implemented as an integral part of the **Oracle** database management system. **Advanced Queues** are represented as regular relational database tables that have been enhanced to support queuing operations like enqueue and dequeue. Messages in queues correspond to rows in a table. The database is then accessed by multiple applications to transfer messages between them, or multiple databases communicate to move messages between their queues (as shown in Figure 3.8).

**AQ** focuses on message management. It can track messages through their entire life-cycle by keeping track of the state of a message at all times (waiting, ready, consumed or expired), and making each message carry its history with it (all the nodes that it has visited and which of the recipients have actually received it). As messages move, they can split, merge and clone, and the causal relationships between them are automatically tracked.

Due to their being regular database tables, **Advanced Queuing** queues can be queried using standard **SQL**. This contrasts with the restricted logging functionality of other MOM products. In addition, applications can optionally retain messages in queues after they have been consumed or propagated to a remote queue. This implies that the local application can carry out queries on its own message history for auditing purposes.

The storage capability provided by **AQ** is, however, insufficient to address the requirements motivated by the discussion of Chapter 2. This is because **SQL** on its own has insufficient expressive power to interpret event histories (see Chapter 5).

## Talarian SmartSockets™

Talarian's **SmartSockets™** publish-subscribe middleware product [Tal98] is a comprehensive message transport that, in addition to all the MOM 'common' feature-set listed earlier, also provides some novel services. **SmartSockets™** provides the application not only with a control API, from where it can send and receive messages, but also a monitoring API, that may be interfaced with either synchronously or asynchronously. In this way, a client application can set itself to receive events pertaining to the state of various internal **SmartSockets™** variables. The product allows dynamic message routing with weighted node connections, where developers can allocate different priority weights to each node that their application can deliver messages to, so that messages are delivered in order of the weights.

Of particular interest are the add-on **SmartModules™** that can be added to the product. One is the **RTie** (Real-Time Inference Engine), instances of which can be placed at various nodes. This allows a developer to specify high-level rules upon which actions are undertaken. The rules allow limited composite event detection and event qualification based on attribute value. Incoming event streams are monitored for the event patterns and one or more rules can be triggered.

Of direct relevance to this discussion are the **RTarchive** and **RTplayback** modules. These provide event/message storage and retrieval for **SmartSockets™** applications.

- **RTarchive** is used to archive or record messages via **SmartSockets™**. All archive messages are saved to a set of files, some of which are indexed by time, message type, or subject. Filters can be set up to archive the data and messages desired. Archiving can be turned on and off by any process in a **SmartSockets™** application.
- **RTplayback** is used to retrieve or playback messages that have been recorded by **RTarchive**. Upon receipt of a request for retrieving messages, **RTplayback** searches for, retrieves, and distributes the selected messages to the terminal, file, or one or more client processes that are part of the same **SmartSockets™** application. **RTplayback** can be used to handle multiple playback requests. There are a number of tasks for which **RTplayback** can be used, including the replay of data faster or slower than real time, the retrieval of historical data for SmartSockets modules, and the analysis of data by exporting it into a file for further examination.

Although the lack of availability of technical documentation on the above modules prevents appraisal of their functionality, marketing literature indicates that these modules attempt to address a subset of the feature-set of the service proposed by this dissertation. The modules can only be thought of as an added local service for a **SmartSockets™** node. They can only be accessed locally, and the retrieval interface is mostly restricted to interactive browsing and manual selection through a graphical user interface.

## 3.5 Conclusion

In summary, this chapter has reviewed existing event messaging solutions. After giving an overview of distributed middleware, it has described the variations in the event model, and then reviewed the most important examples of event-driven systems and message-oriented middleware. Where applicable it highlighted these examples' treatment of event storage.

The above discussion and the preceding one in Chapter 2 demonstrate that existing approaches to event storage are either solely applicable to highly specific application domains, or else are just very limited in scope. To the best knowledge of the author, the storage service outlined in this dissertation goes beyond any existing solution in terms of generic applicability and functionality, and in marked contrast, actually promotes interoperability in between different systems.

# Chapter 4

# A Storage-Enabled Event Infrastructure

This chapter presents a comprehensive event infrastructure called HERALD. This provides event notification, storage and retrieval. The chapter starts by describing a generic and extensible event model, and proceeds to discuss the main features of the HERALD infrastructure. Apart from its novel architectural and technical features based on registration policies, HERALD demonstrates how event storage and retrieval capabilities can be embedded within a messaging or event-driven framework to provide enhanced functionality and performance.

Section 4.1 presents an event model where events are typed, have structure and can form part of inheritance-based event taxonomies. This generic event model supports schema evolution and is supported both within the HERALD infrastructure and within the event repository architecture (see Chapters 5,6).

Section 4.2 gives an overview of the HERALD infrastructure. It defines the conceptual entities within the HERALD event model, describes its concept of registration policies, and illustrates its flexible loosely-coupled approach to dynamic environments.

Event storage and retrieval within HERALD are discussed in Section 4.3. This section describes how *event repositories*, i.e. event storage facilities, can be embedded within HERALD components and gives an insight into how these may be used. This section concludes with a brief look at how an event repository could be employed within message-oriented-middleware solutions.

## 4.1 A generic event model

It is important to first define an event model for use within the event notification and storage infrastructure. This model will be applied within the event-based transport of HERALD as well as internally within the event repository.

## 4.1.1 Event type system

An event is a message that denotes the occurrence of an activity of interest, and belongs to an *event type*. An event type is analogous to an object class, and it encapsulates a number of *parameters* or *attributes*. While an event instance itself usually reflects the occurrence of some activity, its attributes uniquely identify that activity by representing its parameters.

An event occurs at a precisely determined point in time, *a time-point*, and has no duration. In this model, time is assumed to consist of an infinite number of discrete time-points, each of infinitesimally small duration. In practice, however, time-points that are extremely close to each other (within microseconds on modern machines), cannot be distinguished accurately, so events that occur this close to each other need to be distinguished using means alternative to their timestamp.

Each type has associated with it an *event scheme*. This defines and names the parameters associated with the event type, which distinguish each event occurrence. The parameters of an event can be defined in any of the Object Data Management Group (ODMG)'s **Object Model** primitive types [CB97]. It is convenient to adopt this widely used industry standard specification as the base type system, since it also provides a binding from its type-set to the most commonly used language-specific types, including those of **C/C++**, **Java™** and **Smalltalk**. In addition, this makes interfacing with **CORBA** environments and most current commercial databases straightforward. There are only minor differences between the Object Management Group (OMG)'s type system and the ODMG's.

Like classes in an object-oriented environment, event types can *inherit* from a base type event. The base type represents the basic properties of an abstract event, in essence defining the header that is attached to each event instance. These basic properties are the event's creation timestamp as entered at its source, a scalar counter set by the source, and the identity of the source where it originated. Other fields are provided in order to support provision of security. Events are tagged as being of *live*, *past*, or *replay* type. In addition, *past* and *replay* events have a *retrieval* and *replay timestamp* respectively. Furthermore, all events can be *regular* or *compound composite*.

At this point the following definitions must be made:

- A *primitive event* is any event that is generated at some event source in response to some device's activity, and cannot be broken down into any finer-granularity constituent activity. All primitive events are *regular* events.
- A *composite event* is any event that is generated to denote the presence of a particular sequence of events. It is defined as having occurred when the whole event sequence that defined it occurred, this being equivalent to the time of occurrence of the event that concluded the sequence. A composite event still occurs at a particular instant and has no duration. Within HERALD there are two ways of supporting composite events: (1) *regular* events that are associated with a type and have attributes, or (2) *compound composite* events, which are events that encapsulate the constituent *regular* events within them in addition to their own parameters.

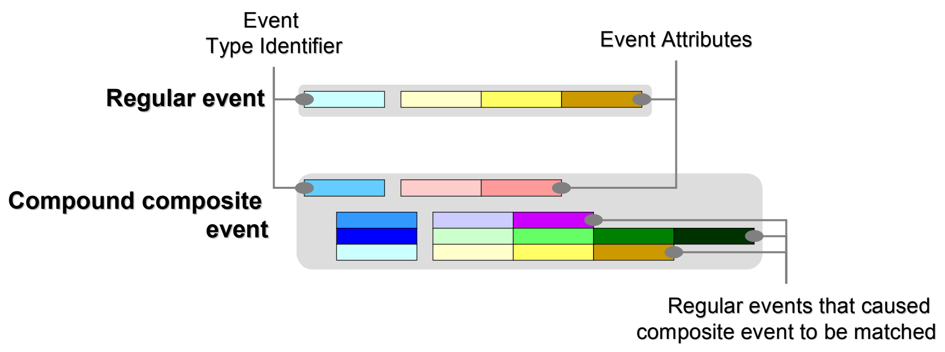

**Figure 4.1**
Distinction between regular and compound composite events

Figure 4.1 illustrates the difference between a regular event and a compound composite event. In the example portrayed, the compound composite event embeds the three regular events (that *might* have been primitive) that defined the matching pattern. In this case, they are shown as being of three distinct different types.

## 4.1.2 Event inheritance

All application-defined event types add parameters to the base event type definition to define instantiable events. Event types with a common semantic meaning can be grouped into an inheritance tree. For example (as illustrated in Figure 4.2), a service that provides information about the location of users can offer the following event type; LocationEvent(Domain, Name, Type, Location). The event type LocationEvent might identify that an entity has changed physical location within a specific application-defined domain. The entity can be, for example, a person or some equipment. There are several technologies enabling tracking of physical movement of people and other entities, with representative examples being the **Active Badge** [HH94] and the **Active Bat** [WJH97] systems, where electronic tags are worn by or attached to the entity being tracked. LocationEvent can then have sub-types inheriting

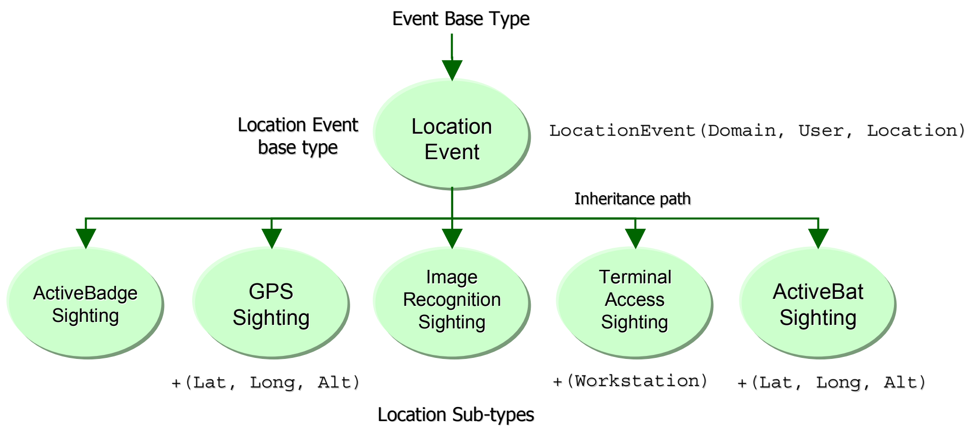

**Figure 4.2**
Example of event type definition

from it that reflect the actual tracking technology being employed, with parameters specific to that technology. These could be ActiveBadge-Sighting, ActiveBat-Sighting, GPS-Sighting, ImageRecognition-Sighting, and TerminalAccess-Sighting. The advantage of being able to specify such an inheritance grouping around the base type LocationEvent is that it becomes possible to carry out operations and queries on the base type that implicitly encompass event instances from all the sub-types. Therefore, while the model supported is analogous to an object-oriented model in an object-oriented environment, it differs in that event types (analogous to object classes) have no behaviour.

## 4.1.3 Event schema

Event type schema themselves have associated meta-data. A scheme is tagged with the name of the event type it represents, a version number in order to support schema evolution, a numeric identifier, the originating component's identifier, and two strings describing the activity denoted by the event.

The first string is a textual description of the activity the event represents, while the second is a structured string with classification information. The keywords defined within the classification information are inherited by any sub-types of the event type being described and may be appended to. The string fields can be employed by event clients while searching for interesting events and their respective sources at *event brokers*. Both the version number and numeric identifier are unique with respect to the context of the event component where the scheme was defined, whose identifier is the third number.

A scheme is *owned* by this event component, details of which are also provided within the scheme meta-data. Although in HERALD any event source can define a new event type and advertise it, it is usually the case that event types are defined at an *event broker*, from where their identification details can be obtained by multiple sources for *adoption* (see Section 4.2.6).

The rest of the scheme structure defines the event type's parameters and inheritance details. Each parameter definition also has attached to it a textual description of the attribute represented by the parameter.

When an event is dispatched to an event client, its scheme's compound identifier is attached to the event instance's parameters.

## 4.1.4 Event evolution

In a dynamic and reactive environment like a messaging infrastructure, where the event repository is an important component, application components can be expected to change and evolve independently of each other. Therefore, the event model supports schema evolution through schema versioning. A new evolution of a event type's scheme need not re-define the whole inheritance tree from the base type event downwards. It need only re-define the tree from the first node altered onwards. This is illustrated in Figure 4.3. In this example, one can see the effect of defining

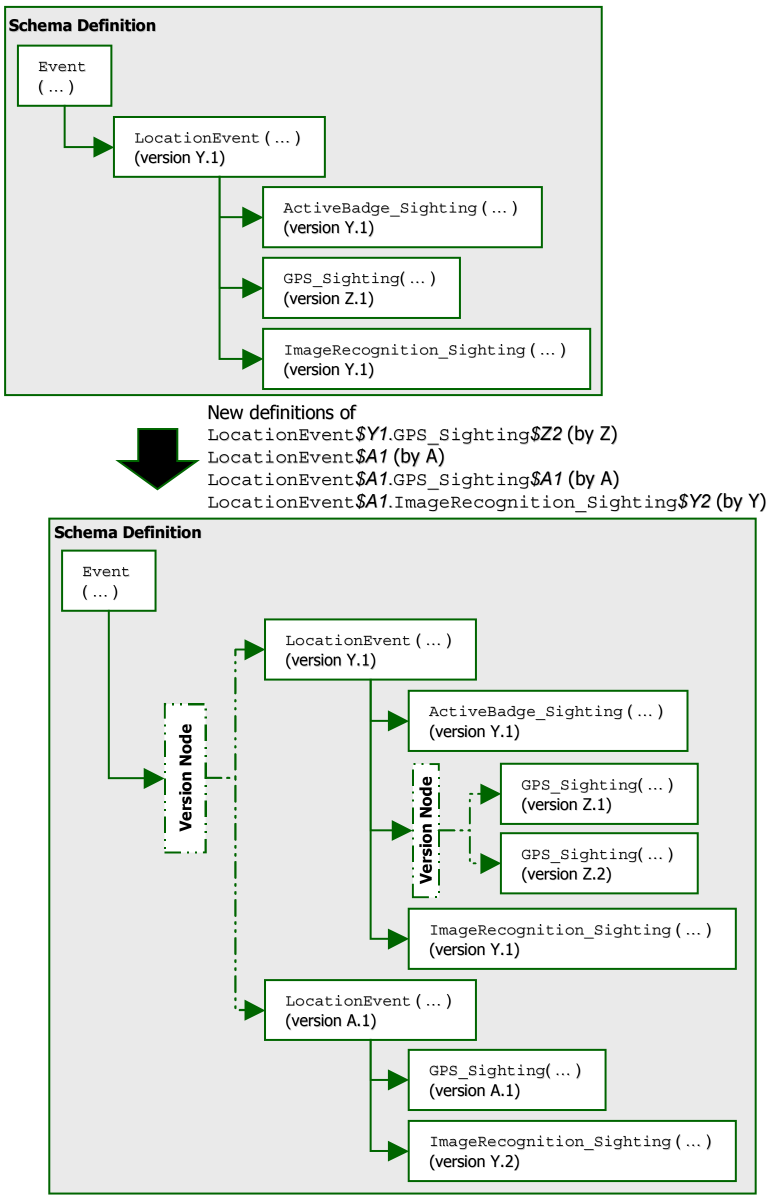
Schema Definition
Event ( … )
LocationEvent ( … ) (version Y.1)
ActiveBadge_Sighting ( … ) (version Y.1)
GPS_Sighting( … ) (version Z.1)
ImageRecognition_Sighting ( … ) (version Y.1)
New definitions of
LocationEvent$Y1.GPS_Sighting$Z2 (by Z)
LocationEvent$A1 (by A)
LocationEvent$A1.GPS_Sighting$A1 (by A)
LocationEvent$A1.ImageRecognition_Sighting$Y2 (by Y)
Schema Definition
Event ( … )
Version Node
LocationEvent ( … ) (version Y.1)
ActiveBadge_Sighting ( … ) (version Y.1)
Version Node
GPS_Sighting( … ) (version Z.1)
GPS_Sighting( … ) (version Z.2)
ImageRecognition_Sighting ( … ) (version Y.1)
LocationEvent ( … ) (version A.1)
GPS_Sighting( … ) (version A.1)
ImageRecognition_Sighting ( … ) (version Y.2)

**Figure 4.3**
An example of schema evolution

four new versions of existing event types. The new type definitions do not overwrite the previous versions. Rather, they are attached to the existing tree, adding an alternative dimension to it. Each entry is identified through the version number given to it by the component that defined it, and needs to be fully qualified with respect to its position within the evolving schema tree. Therefore, LocationEvent$*A1* refers to the definition version 1 of LocationEvent made at component A.

## 4.2 Overview of HERALD

The HERALD event system is based on the revised version of the **Cambridge Event Architecture** (**CEA**) [BBHM95] described by Bates et al. in [BBMS98], and endorses the *publish-register-notify* programming paradigm.

HERALD differs from **CEA** in that it does not employ any interface definition language to describe the event types supported at sources. **CEA** uses these definitions to construct event stubs that are linked in at compile time with the event client components. This model imposes statically defining and publishing an event source's events at compilation time, implying that if an event source is subsequently modified, the clients that were communicating with it need to be recompiled. Instead, in order to support loose-coupling, HERALD endorses a purely reflective interactive model, where components discover and learn about each other's capabilities and events at run-time as required. This is not entirely novel in concept, as the **CORBA 2.0** [OMG99] specification provides for both a static and a dynamic invocation interface. However, few **ORB** implementations support the dynamic interface, thus restricting the developer to build a closed system.

In a distributed system built using HERALD, each event-aware application unit is known as an event component. There are two primary types of conceptual components, *event source* and *event client* components. Conceptually, events are structured objects, and flow from event sources to consuming event clients. *Sources* can be wrappers around an application or device, and based upon the monitoring of some property or activity within that application/device, generate events pertaining to it. *Event clients* register interest in events directly with *event sources* and are sent *event notifications* when events matching their *registrations* occur at that source. This structure is shown in Figure 4.4.

In practice, the functionality represented by either component type is provided through a package of class libraries (see Section 7.1.1), of which **EventClient** and

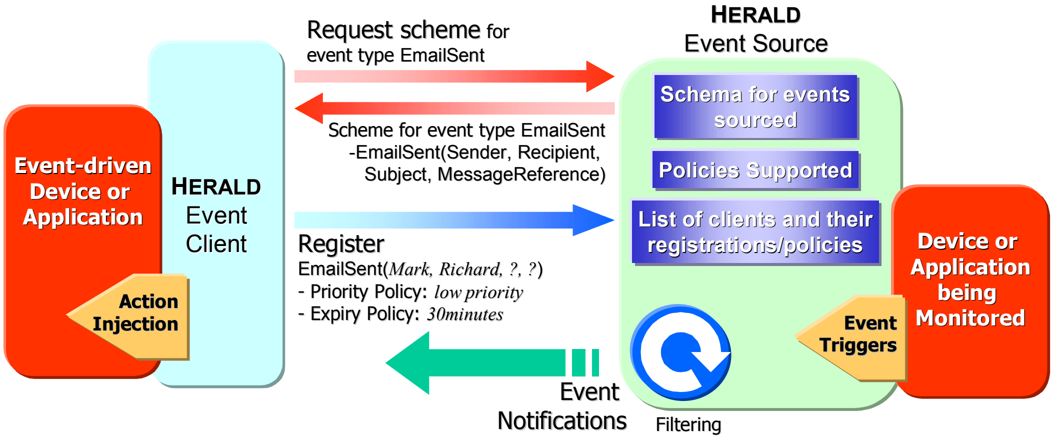
Request scheme for event type EmailSent
HERALD Event Source
Schema for events sourced
Policies Supported
List of clients and their registrations/policies
Scheme for event type EmailSent -EmailSent(Sender, Recipient, Subject, MessageReference)
Event-driven Device or Application
HERALD Event Client
Action Injection
Register
EmailSent(Mark, Richard, ?, ?)
- Priority Policy: low priority
- Expiry Policy: 30minutes
Device or Application being Monitored
Event Triggers
Event Notifications
Filtering

**Figure 4.4**
A simple registration sequence in HERALD

**EventSource** embody the functionality of a HERALD event client and a HERALD event source respectively. These libraries abstract away most of the detail pertaining to event registration, filtering, event queuing, communication, fault tolerance, event persistence, reliability and security. Application writers can wrap event functionality around their application units or legacy interfaces very rapidly without concerning themselves with the underlying middleware intricacies. A hardware monitor can therefore have an event source written around it that internally communicates with the proprietary device interface, but only 'exports' events to the outside world.

The above discussion used the work 'conceptually' because a component, or any independent software entity, need not necessarily map to a *source* or a *client*. In fact, it can be both a source and client of events concurrently with respect to different components. By integrating both source and client functionality, it is possible to built specialised transforming components like *federators*, *gateways*, *storage modules*, and *event brokers*. These are analogous to the concept of *mediator* components within the **CEA** framework. More detail on these services will be given shortly.

In HERALD, communication between event components is asynchronous, with event propagation being decoupled from the execution at the source and the client. When one or more events are received at a client, the HERALD **Event Client** module passes them into the application through an application-specified object handler. Different object handlers can be invoked according to the type of the notification received. Likewise, although they do not have to, an application developer can tap into and monitor inter-component control messages. Examples of the latter are event registrations and enquiries as to what capabilities are available at an event source.

Components are denoted by a component identifier, which is a structure that contains a unique identifier, the component's network and communication details, and, if applicable, the component's identifying digital public key.

## 4.2.1 Registration templates and policies

HERALD clients register interest in event types with one or more event sources. In order to carry out such a registration, the client must first locate the event source it wants to receive notifications from. It can do this either by knowing the location of the source on the network in advance (i.e. the location being hard-coded in the application code). A more likely scenario in a dynamic environment is that a client queries an *event broker* in order to search for appropriate event sources. It can carry out this search by providing the broker with details of the event types it is interested in, upon which the broker will provide it with the location and naming details of relevant event sources. The client can then enquire of each source its capabilities and if required request the schema of the events that source can supply. It then submits an event registration to the source.

In the **CEA** model, a registration consists of an *event template*. This is an event instance with fields for exact match filled in and those for wild card match expressed as variables. Filtering is therefore carried out at source, and the client is notified only of those events that are of interest to it, in that they are not only of the type that it wanted to monitor for, but also match one or more parameter values. This implies additional complexity at the source, as it must now keep track of all the registrations pertaining to its clients (and effectively track all those clients) and carry out template matching for each event that is generated. However, this approach reduces network bandwidth, allows for simpler clients, and enables better control and management on service provision, in line with current Internet trends. Registration of interest in a super-type event, as defined within the event taxonomy of that source, can be extended to cover all the sub-types of that super-type.

HERALD enhances this registration model by extending it with *registration policies*. In HERALD a registration consists of two segments, the first being the event template as described above, while the second part being a list of one or more registration policies. Policies qualify the registration, in that they alter the way in which the registration is to be carried out or its scope. There are several policies defined in HERALD, and the model is flexible enough to support additional ones being attached at a later stage. It is possible for a client to carry out two registrations for the same event type with different parameters and different qualifying policies. The main categories of registration policies are:

- *Priority* – A source can support multiple delivery queues with different priorities. If this feature is enabled, a client can differentiate its registrations based on how important it deems having those events delivered to it rapidly. The number of priority levels and the default priority is customisable at the source, and can be enquired about by clients. For example, a event client component that monitors network components can ask to have emergency 'high load' events sent to it with very high priority, while regular service-level data is dispatched at lower priority.
- *Frequency* – Frequency policies represent constraints on the volume of event notifications that a client wants to receive with respect to its registrations. For example, a client can request that it not be sent more than one identical notification within a specified amount of time. This is particularly suitable for recurrent error notifications such as those generated by the file system filling up. Two hundred identical error events are unlikely to be very useful.
- *Expiry* – A client can attach a policy to its registration setting an explicit expiry condition on the lifespan of the registration. Examples of such conditions are that the registration is only to last for a specified number of event notifications or for a length of time from its acceptance at the source.
- *Storage* – Storage policies are of importance in this context as they are available only if the event source encapsulates an event repository and provides event storage and retrieval capabilities. Apart from conventional message queuing *store-and-forward* capabilities, clients can demand that their notifications be dispatched in batch every specified amount of time, or that notifications be retained for them until they send a trigger requesting dispatch to a specific location. The latter policy addresses the issue of mobile clients, or disconnected clients on dynamic IP addresses (like users on dial-up networking connections). Other policies enable retrieval and dispatch of past events matching the template submitted, or

even fully fledged temporal queries on the events that the source has generated in the past. How far back in time the source retains events is a property set by the application writer responsible for implementing the source. Persistence capabilities are described in greater depth in Section 4.3.

- *Reliability and fault-tolerance* – In analogy to how in key message-oriented-middleware products one can request guarantees on delivery, event clients can attach reliability policies to their registrations. Fault-tolerance policies then specify how the source should behave in error situations. If a connection-based transport like TCP/IP is used for event dispatch, the source can tell when delivery fails. In this case, one option is to assume that the client is dead and to purge all its registrations, while another policy guarantees delivery by retrying delivery at regular intervals until an acknowledgement is received from the client. Event clients implement a session protocol where they attach a session identifier (reflecting an execution run) to communications with sources. This allows a source to recognize when a client has crashed and been restarted. A *heartbeat* protocol service can be requested, where the source sends a regular heartbeat event to a client to signal its continued uptime and verify communications. Alternatively, the source can provide a regular heartbeat signal on a multicast group address that clients can listen on.
- *Security* – There are two aspects relevant to this area; authentication of clients/sources, and security of transmission. HERALD supports the concept of clients attaching a public key as a policy parameter, which is then used by the source to transmit encrypted events. If the key is issued by a trusted third party, then it also serves to authenticate the client. It is envisaged that a more elaborate and comprehensive event security model, like **OASIS** [Hay96], could also be applied.

In contrast to earlier event-driven environments, which compromise between functionality and performance, HERALD allows each source component writer complete freedom to enable as many or as few policy modules as wished. The core source functionality consists of the basic **CEA** registration-notification capability, which is ideal for a lightweight but functional event notification service. On top of this, one can then enable the modules that provide multiple priority queues, security, reliability, and persistence (which launches an event repository within the source), and choose the individual policies to support. The compromise lies in a degradation of performance and an increase in storage requirements, against additional functional service to event clients. A further enhancement on this model would be differentiation of service by client, undoubtedly a requirement in an open commercial environment. In this case, while some clients would see a core lightweight event source, other more favoured clients (qualified as such through their presentation of an appropriate identifying certificate) will be able to enjoy greater functionality.

Furthermore, a source component can be written to monitor its own load and gracefully reduce its service by selectively disabling its feature-set dynamically. As this affects the registration policies supported, clients can then be notified that the policies they attached to their registrations can no longer be supported.

Activation of each module effectively enables a number of policies, and these are advertised in much the same way as the event types the source can supply, i.e. through brokers or directly to clients.

An event client can also be customised along similar lines. It can also embed an event repository, a federator module (see Section 4.2.5) and use a sliding event observation window to enforce global ordering of events (see also Sections 4.2.4, 5.2.6, 6.3.1).

## 4.2.2 Component communication

Communication between HERALD components is decoupled from the application execution at either side, and is connection-less. Messages are placed on a queue at one side, and then dispatched and placed in a queue on the receiving side. They are then processed, and if necessary, an acknowledgement is sent back in the same asynchronous manner. Control messages sent by the client to the source are always acknowledged, so that the client knows that its request has been received. In addition, the source can return information pertinent to the nature of the control message within the acknowledgement. An example of this is a client request to be sent the scheme of an event type, or to carry out a registration. Clients label their control requests according to their own individual labelling criteria, and the label is returned by the source within the acknowledgement.

The event and message data is sent over the network in a language and platform independent format, allowing easy portability of HERALD across heterogeneous environments.

Given the individual settings pertaining to event notification (e.g. event types, parameter templates, registration policies) that HERALD allows clients to define, most communications within the HERALD model are unicast-based. There are inherent problems with employing multicast for event dispatch. Due to the potential of a client receiving thousands of unwanted events per second, in multicast scenarios clients have to carry out local filtering and in general run on a more powerful platform. In addition, multicast across wide-area networks is not always possible due to technical restrictions, and there are a limited number of multicast-addresses. Therefore, conflicts need to be managed. Reliability and security become difficult to address and require complex solutions (often requiring resorting back to unicast).

## 4.2.3 Dynamic nature

HERALD was designed for deployment in loosely coupled *dynamic environments* like the Internet. It is assumed that application components are developed independently of each other, and are likely to be under different management. An implication of this is that components can be created and destroyed without notice, and will evolve over time. Component evolution spans basic core technologies like the event transport itself and the nature of the event data available. Just as any software technology evolves and matures over time through new versions, likewise, one can assume that newer components may deploy newer versions of HERALD. These may make available new functionality and support a different instruction-set in component communication.

In a separate issue, the event types available at a source may evolve over time as the underlying triggering conditions change. For example, an event type CashWithdrawn(ATM, Customer), reflecting a bank's Automated Teller Machine network, may evolve to CashWithdrawn(ATM, ATM_Owner, Customer) when the bank sets up an agreement with other banks to allow its customers to use their ATM networks. Alternatively, a LineDrawn event in a shared whiteboard application may be changed in a newer version to a different set of parameters to reflect whether the drawing of the line was via a mouse or a handheld pen.

In a dynamic environment, it is not possible to recompile or even restart remote components when these changes occur, or otherwise assume that they will be modified to cope with the changes. HERALD addresses these issues through the provision of a core set of immutable inter-component messages providing *reflection* and *versioning* services. A component can therefore enquire as to the capabilities of any other component, the event data it provides, and the version of the event interface it supports. Ideally, newer interfaces should support older versions but do not have to do so. For this reason, the schema defining the parameters attached to an event type are versioned, and the version identifier is constructed in relation to the source that defined each scheme in the first instance.

Similarly, functional services that may be available at an event source and are represented by policies are also versioned. Likewise, all constructs that are transmitted in between components, like events, control messages, acknowledgements and schema, are tagged with the version number of the HERALD event transport libraries that they were created by.

An event client may ask for the scheme of an event type from a source, and it may request a list of the policies (be they related to registration, security, reliability, etc.) supported by that source. It may also enquire as to whether any particular policy is supported by that event source, and the reply from the source will list the versions of the policy functionality supported if applicable.

### 4.2.4 Causality, distributed time, and timestamps

In any distributed event-based system, there are two concerns related to time:

- *Lack of global time* – Each machine has its own clock, which may drift at varying rates.
- *Network delay* – Events sent between machines will incur a delay dependent on machine and network loads.

The theoretical problems of *timestamping* events and synchronizing clocks across distributed systems are well known. Lamport first examined the problem of distributed clocks and imposing global ordering of events in [Lam78]. However, the full ramifications associated with wide-area scaling are not yet fully understood.

Amongst the distinguishing characteristics of a loosely coupled system is the variable and potentially vast number of components in it. Because of this, it is infeasible to employ several mechanisms that could be used to support event notification in a tightly coupled application. *Vector clocks* [Fid91, Mat88] or the more complex *matrix clocks* [FM82], which piggyback on each message exchanged between the event components a vector timestamp that aids identification of causally related events, cannot be used because the timestamp's size is linear in the total number of components in the system.

Applications therefore have to accommodate approximate representations of time, such as assuming the existence of a global clock even though such an assumption may result in inconsistent observations in different frames of reference. A number of approaches can reduce the negative implications of this assumption, and are surveyed comprehensively in Dietz [Die96]. Commonly, clocks are synchronised within a local area network using the **Network Time Protocol** (**NTP**) [Mil91] that achieves an accuracy of below 100*ms* even in wide-area networks. This can be coupled with data from the **Global Positioning System** (**GPS**) [AL95], which transmits **Universal Time Code** (**UTC**) signals that have an accuracy of between 0.1 – 10*ms*. By employing the notion of an *approximated time base* [Kop92], this implies that all events that occur outside of 20*ms* from each other can be ordered.

Because of these issues, all HERALD timestamps are tagged with the identity of the component that placed them as well as an *event sequence number* attached by that component. The timestamps themselves reflect *real time*, and consist of two long values that denote the number of microseconds since 1st January 1970.

These issues are discussed further in Section 5.2.6.

## 4.2.5 Federators, action-injection and composition

In [BBMS98], Bates et al. describe how event-based components can have an interface for action injection. This allows a client of a component to invoke methods that *inject actions* into the component. By authoring a glue component, known as a *federator*, the actions to take upon event receipt can be specified through a set of declarative rules.

HERALD enhances this through a flexible 'building block' model. There are three ways how this functionality can be achieved in HERALD:

- In the basic HERALD model, an event client is effectively a wrapper around a non-active component that exports a control interface. An event handler object is then written by the application writer, and this, upon being passed event notifications pertaining to a specific registration at a source, calls the appropriate methods on the underlying component (thus injecting actions into it).

- Alternatively, the application writer can use a pre-defined federator handler, and pass all the event notifications to it. Rule-based action rules, which determine what actions it should undertake, i.e. what component methods it can invoke, can then be issued to the federator. The federator applies its rule-based logic to whatever new events are received, and encapsulates a composite-event engine (see below).
- In the third and most powerful variant, the event-client/federator and the target component can be disjoint software entities, running on the same or on different platforms. It is also possible to have one federator triggering actions in multiple components. Communication is carried out through a conventional RPC-based interface between the federator and a lightweight HERALD injection interface wrapper around the target component.

Figure 4.5 illustrates the configuration applied by the second variant. The third would be represented similarly. As described in [BBMS98], rules take the following format:

rule <*name*> <*Event Expression*>{*Actions*}

where the event expression can consist of a composite sequence of events. Actions can therefore be triggered after the detection of a complex pattern of events. Figure 4.6 illustrates how a federator can be used to compose a shared whiteboard cooperative application. Chapter 5 discusses the grammar of composite event specification in detail.

Federator rules call upon developer-supplied methods. By writing a method that triggers new events and using it to drive a HERALD event source interface, an event *mediator* can be built.

It is therefore possible to compose components that register interest in events

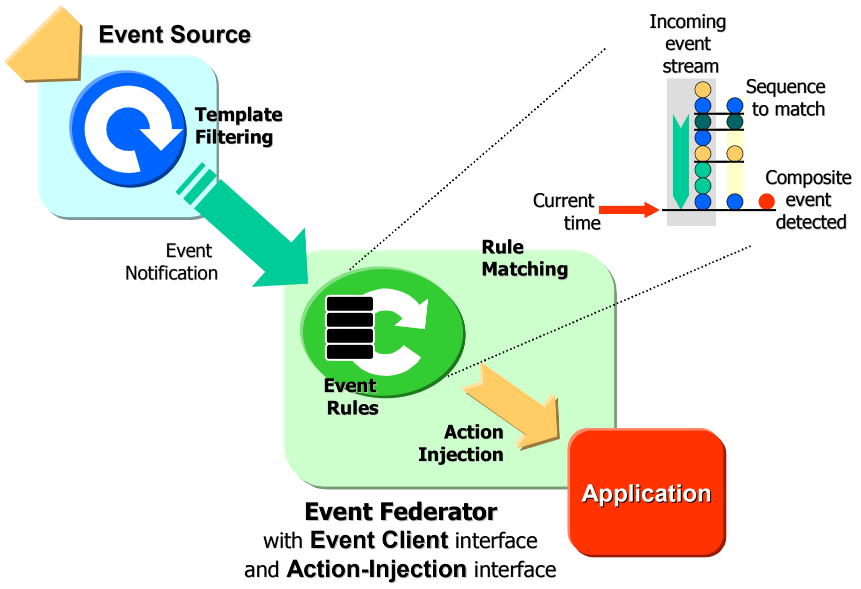

**Figure 4.5**
An event federator that applies rule-based logic to inject actions inside components

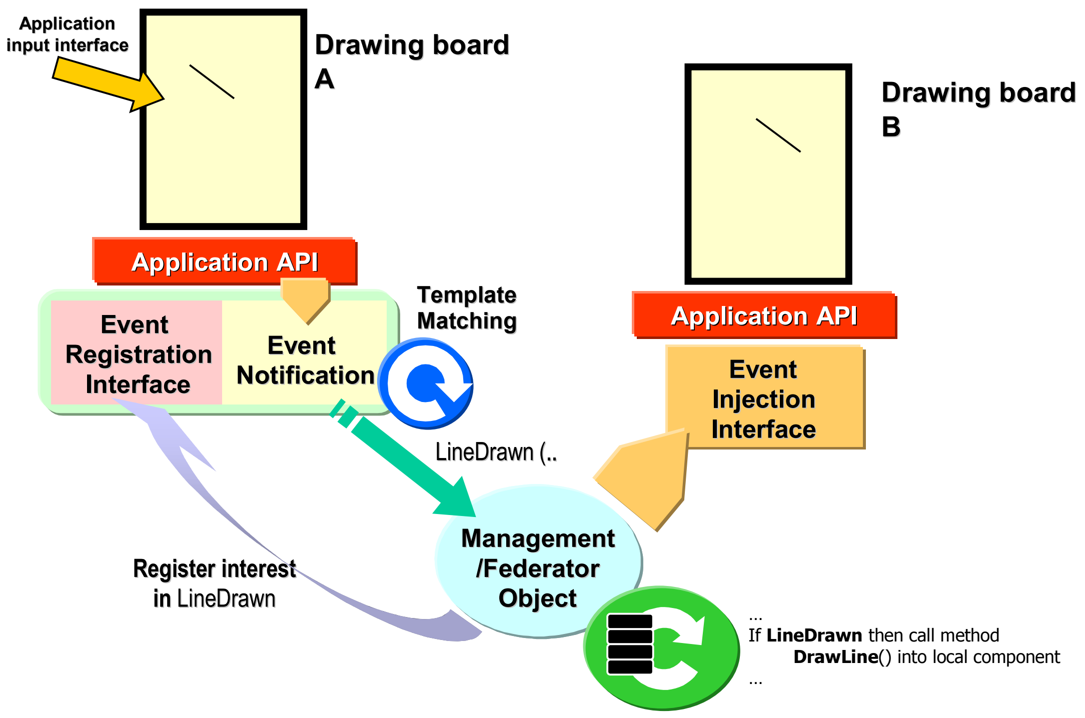
Application input interface
Drawing board A
Drawing board B
Application API
Event Registration Interface
Event Notification
Template Matching
Application API
Event Injection Interface
LineDrawn (..
Management /Federator Object
Register interest in LineDrawn
...
If LineDrawn then call method DrawLine() into local component
...

**Figure 4.6**
Using a federator to compose a shared whiteboard application

from remote sources, apply some rule-based logic upon receipt of notification, and generate new events of their own. In HERALD, as in **CEA**, these are called *mediators*. A mediator has both an event client and an event source interface. It registers interest in events from remote sources, applies some computation relating to the events that is notified of, and generates events of its own. A *composite mediator* issues new events that are triggered upon detection of a sequence of events, and whose parameters reflect the parameters of the original constituent events. Composition of events from *primitive events* into *composite events* is discussed at length in Chapter 5. A *transforming mediator* can apply some logic to augment or translate event data. A *gateway service mediator* works in similar fashion to map events from one event taxonomy to another, and act as a gateway between event domains. These components can all embed event repositories (see Section 4.3.3).

## 4.2.6 Event brokers

An *event broker* is analogous to a trader. Event sources publish details of the events they have available with it, and clients can interrogate it to locate appropriate event sources. Note that this differs from a *Message Broker* as frequently defined in MOM middleware, where publishers pass their messages on to the broker, and the broker then forwards the events to the subscribers. As shown in Figure 4.7, in HERALD, the broker acts as a repository for the event taxonomies available at its local event sources, and its use is in bringing event clients and sources together, rather than transparently decoupling their identities and acting as a go between for event dispatch.

As described in Section 4.1.3, event types in HERALD are defined by schema which themselves have associated meta-data. Although in HERALD any event source can define a new event type (with its scheme) and advertise it, it is usually the case that event types are defined at an event broker, from where their identification details can be obtained by sources for *adoption*. *Adoption* means that an event source downloads the scheme for an event type from an event broker, and generates its events according to the scheme. Should it be asked to provide the scheme for its events to an event client, the source will forward the scheme it had obtained in the first instance from the broker. If the scheme subsequently *evolves* at the broker (for example to contain additional parameters), the event source does not have to know or update its events. It can continue serving events under the old scheme, as both will be distinguishable by their version identifiers.

Brokers can communicate and federate their taxonomy information. [BBMS98] describes a number of ways in which event brokers can be federated.

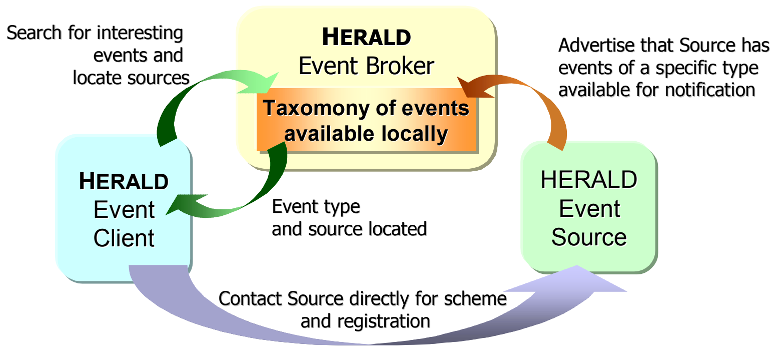
Search for interesting events and locate sources
HERALD Event Broker
Taxomony of events available locally
Advertise that Source has events of a specific type available for notification
HERALD Event Client
Event type and source located
HERALD Event Source
Contact Source directly for scheme and registration

**Figure 4.7**
Searching for events and sources at a Broker

## 4.3 Applying event storage and retrieval

This section describes how event storage and retrieval fits in within the infrastructure described.

Current event-driven and message-oriented-middleware frameworks restrict event-clients to acquire only live information. An event client will only receive events that occur after it has established its connection to the event source or notification service and submitted some sort of registration of interest.

As described in Chapter 3, persistence in most of these systems is limited to having persistent message queues that can survive system failure, so that upon restart, the queue can be reinstated. The system can then proceed with dispatching those messages that were outstanding on the dispatch queue before failure. What an event repository enables, however, is storage of part or all of the message history generated at the message source. This enables several novel applications not previously possible in messaging applications, some of which were illustrated in Chapter 2.

The advantages of storage are more clearly outlined in some applications than in others. These advantages are more obvious in an environment where the messages or events are not intended for a specific target but are available for general consumption. Retaining highly specific peer-to-peer application messages in a tightly coupled environment is not as useful as retaining events in loosely-coupled distributed systems where event consuming-clients can come and go dynamically, and sources make no implicit assumptions about their clients.

Event storage can be employed in two ways within an event infrastructure:

- *within event components where its use is implicit,*
  By embedding an event repository within them for their interval use, some core components can provide enhanced services like buffering, store-and-forward, state querying, and history retrieval.
- *at dedicated storage nodes where it provides an explicit service.*
  In the same way as there can be a *mapping mediator* that modifies messages, or a *composite-event mediator* that collect fine-granularity events and generates composite events, there can be a *storage mediator*. This component is a dedicated storage node whose sole purpose is to capture and store events. Other components can then query it to retrieve or replay event streams.

A comprehensive description of the feature-set of an event repository and its interface are not given here, as that is the subject matter of Chapters 5 and 6.

## 4.3.1 Embedding of storage capability within core modules

The HERALD core modules that provide the functionality of an *event s*ource and of an *event client* can both embed an event repository. Both modules allow an application developer to activate retaining of event histories and specify parameters that specify for how long, and how many, events are to be retained. Chapter 5 describes a comprehensive query and control interface that allows extensive interaction with a repository, and the HERALD modules use this interactive capability internally while exporting an API-based interface to the application developer.

### Event client module

In the case of a HERALD client module (see Figure 4.8), the events retained will be events that have been generated elsewhere and received by the client because of registrations it has carried out with remote HERALD event sources. The application can specify which of the incoming event streams must be retained. Therefore, the events stored can be of different types and originate from different event sources.

By default, these events will be stored and time-stamped with their storage time, although their creation time, applied by their sources before notification, is retained, and can be utilised for querying. The former may not always be consistent with global ordering across the different remote sources. As reflected in Section 4.2.4, apart from the problem of remote HERALD sources having system clocks that are not in perfect synchronisation with each other causing shifted relative ordering, network delays in transmission can reflect a different temporal separation. While the latter can usually be ignored (except where high-precision replay is required), the former can be troublesome. Nevertheless, if the application wants to retain a history of the events as it received them, then the storage timestamp represents an accurate reflection of that view.

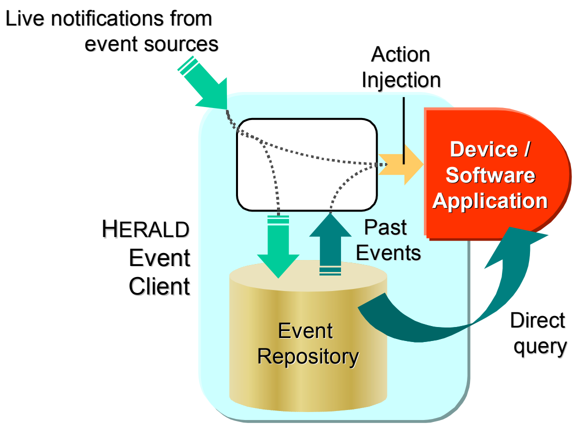

**Figure 4.8**
Embedding an event repository within an event client

The incoming events can also be grouped in *event sessions* when stored. Sessions are described in Section 5.2.5, and are a means of associating event instances of the same and of different types into meaningful groupings. An event instance may belong to one or more sessions, and retrieval and replay queries can be carried out by session rather than by event type.

## Event source module

On the other hand, only locally sourced events are captured within a HERALD event source when this embeds an event repository (Figure 4.9). This enables the source to retain its own history of the events that it has generated over time. Time-stamping and ordering in the event repository is not an issue here, as the events are being captured locally. In this case, as well, since the source may be able to generate several event types, the application determines which are stored and notified, and which are not retained beyond notification to interested HERALD clients. As with HERALD client module storage, the outgoing events to be retained can be associated through event sessions. As described in Section 4.2.1, event storage and retrieval within event sources is an implicit requirement for a number of HERALD's more powerful features. A number of policies can only be made available if a

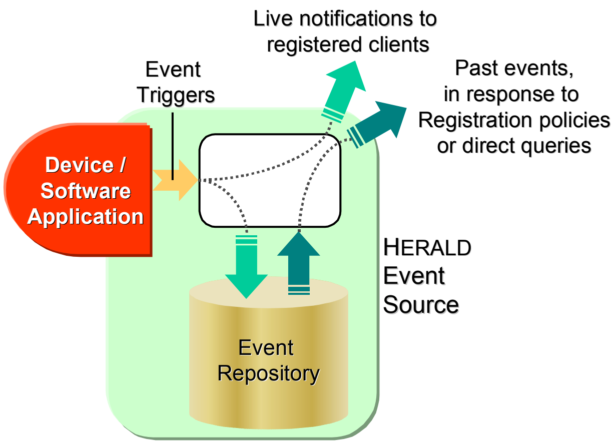
Live notifications to registered clients
Event Triggers
Past events, in response to Registration policies or direct queries
Device / Software Application
HERALD Event Source
Event Repository

**Figure 4.9**
Embedding an event repository within an event source

source contains an embedded event repository service. By requesting these policies, the client components cause the source module to interact with its event repository.

## 4.3.2 Applications of event histories

This section discusses some of the capabilities enabled by event storage within HERALD. Section 4.3.2 then illustrates where these capabilities are applicable.

### Retrieval and replay of past events

A HERALD event source equipped with an event repository can allow clients to be more selective in their queries as regards its history of the events it has generated. An event repository goes beyond being a data store or buffer of events, and supports a dedicated retrieval interface (see Chapter 5 for a discussion of TEQL).

HERALD clients can demand that they be forwarded event notifications, matching their registrations, which occurred in the past. Policy parameters determine whether this is done in batch or sequentially. These events are tagged as *past events*, and have both a transmission timestamp and the original creation timestamp.

A HERALD event client can also request replay of events from a storage-enabled HERALD source. In order to do this, the client can provide two time values to demark the interval of activity to replay from. Alternatively, it can attach a TEQL query to its 'replay' request. This enables a client to detail the past events it wants to know about in terms not only of their content, as in their parameters, but also in terms of when they occurred with respect to time, or with respect to occurrences of other events. This 'single shot' registration, in that it is processed, handled and then deleted, processes the event history to locate the events that satisfy the query, and then initiates a replay at the same temporal separation as the original events occurred. The client receives the relevant event streams as if they were live data. *Replayed events*, however, are tagged as such, and have both transmission and creation timestamps.

### Initialisation and reconstruction of state

Event information often reflects a *change of state* of some kind. In a location monitoring application, this could be a PersonMovement event, which occurs whenever an individual walks from one location to another. Alternatively, consider an event reflecting the share price of a company reaching a certain threshold. These events were triggered by a change, and therefore any client application that starts up and registers interest in the change will have no idea of the *current value* of the state being monitored until a change in it takes place (unless it can obtain the information through some bespoke communication).

If the event denotes a change to a new value, being able to retrieve the last instance of the event that occurred before the registration was affected, enables the client to initialise its data until it is notified of a further change. This does not merely require the event source retaining the last occurring event instance of any event type, since a HERALD client would require the last notification that matches its *registration template*. Therefore, different clients would require different past instances.

In other cases, re-initialisation of state requires more than just one past instance. For example, an application might require all events since a particular time-point. If the application logic is based around processing external events, obtaining all the events again can enable it to re-instantiate its state following failure. Alternatively, the event client embedded within the application could have captured and retained all the input event streams, whereupon the application could re-apply them to itself in the same sequence as it received them in the first instance. Obviously, this requires dedicated fault-tolerant logic within the application.

In another scenario, individuals that join a cooperative work session late can have their cooperative application's event client request all past events from the event sources attached to their colleagues' applications. Their application would then be able to initialise itself to a state consistent with their colleagues'.

## Encapsulation of functionality

When an event source is interrogated about the policies it supports, it can advertise that it can provide information on past events of a particular type. Taking this further, an application-writer can decide to advertise that a source can provide past events that in fact it does not even provide live instances of. This can be achieved because the source can be made to search its local event history and locate *composite events* that consist of patterns of the events stored. Composite events are typed and structured just like primitive events, and can therefore be forwarded to clients as either normal event messages (also known as *regular* composite events), or as *compound* composite events, which also encapsulate the constituent regular events that made them up.

## Enhanced support for mobile and disconnected clients

Conventional store-and-forward capability can be optimised by the source not having to retain and buffer separately event streams for each of its clients. Instead, if the data is to be forwarded upon a trigger or at a time value, it can be retrieved from the local event store encapsulated within the event source, just prior to its being dispatched.

In this regard, HERALD allows clients to set up buffering policies where either all their notifications are dispatched periodically in batch (e.g. every fifteen minutes), or else their notifications are dispatched upon receipt of a trigger message by the client. When a client first carries out such a policy-qualified registration, it is issued with an authorising certificate contained in the acknowledgement returned to it by the source. The trigger it then sends to obtain its data must contain this certificate in order to identify itself, and not only indicates that it is appropriate for the source to dispatch the events, but also specifies where to dispatch them to. This feature addresses the problem of disconnected and mobile clients. These occur in three scenarios:

- The first is that of event clients that are in effect mobile agents or mobile applications, which retain state and carry on execution as they relocate from location to location.

- The second category is that of applications on physically mobile platforms – such as applications running on handheld devices, embedded ubiquitous computers, or notebook and laptop computers. When these reconnect to a fixed network, they often do so at different endpoints and therefore obtain a different location identifier (e.g. IP address). Unless low-level transports transparently address this issue of mobility, as proposed in the **Mobile IP** [IET99] draft, it needs to be addressed at middleware or application level.
- The third category is conceptually similar, in that it consists of all those millions of home users that connect to the Internet on a non-permanent basis through a telephone line to an Internet Service Provider (ISP), and are issued with a different IP address each time.

## 4.3.3 Storage-enabled event components

Using the storage capabilities inbuilt within HERALD's event source and event client modules, a number of storage-enabled event components can be built.

### Monitoring Event Source

A *monitoring event source* is an event source that monitors some hardware or software, both here termed a *device*, and generates events to reflect activity within the device. This is the most common component in any event infrastructure. Often, the event source is structured like a wrapper around the entity being monitored, and acts like a bridge between its proprietary interface and the rest of the event-driven system. An example is the HERALD-based **Active Badge Event Source** that is illustrated in Section 7.3. This is attached to the software driver that interacts with the networking hardware of the **Active Badge** sensor network. While other applications that wish to receive badge sightings have to implement the complex interface of this software driver, HERALD client components need only connect to the **Active Badge Event Source** and register interest in sighting or movement events. Another example is an email reader event source that monitors the receiving and sending of email, and generates informative events that reflect information about the emails sent.

By embedding an event repository within the source module, the event source can retain, for some time at least, a history of the activity that has taken place in the application it is monitoring. This has three primary purposes:

- *Initialisation of state.* As described in Section 4.3.2, remote clients can request to receive details of past device activity that they may require in order to initialise their data structures, in anticipation of future notifications.
- *Analysis of device performance and activity.* An application attached to the event source, or alternatively a remote client, can use the embedded history to analyse activity within the device being monitored. It can review the frequency with which events have been generated, locate recurring sequences of events, and summaries of activity that may be browsed by a user can be generated. The data may be used by service management software to modify the functionality of the device, or to start alternative instances of it.

- *Enhanced service provision.* Remote clients can specify that they only want updates on the device's activity at periodic intervals, or that they do not want to be notified of repetitive events.

## Mediator components

HERALD mediators have both an event client as well as an event source interface. They are notified of events, process them, and generate new events. For this reason, they can store either the incoming events on the client side, the generated events on the source side, or both. If both their client and source modules activate event storage, they share the same event repository. This is illustrated in Figure 4.10.

*Gateway mediators* are like proxy event sources, in that they register interest in events from external domains, and then forward them to interested parties within their local domain. By retaining events within an embedded event repository, they provide their local clients with the same storage service as they would have if they communicated directly with the remote event sources.

*Composite* and *transforming mediators* consume events and generate new events of different types. Composite mediators parse the incoming event streams to look for predefined event sequences. When they detect such a sequence they then generate a composite event, which to all intents and purposes can be identical to a normal primitive event. Transforming mediators apply some application specific logic to translate events from an event taxonomy to another. Both types of mediator can embed an event repository, and utilise their event histories to fine-tune their behaviour as well as enable historic event retrieval services.

## Storage mediators

A storage mediator is a dedicated storage component. The aim of a storage mediator is to collect and store events. It registers interest in events and retains them once they are notified to it. It can be dedicated to one application, in which case it is likely to be attached to such an application. The application then uses TEQL to access the stored

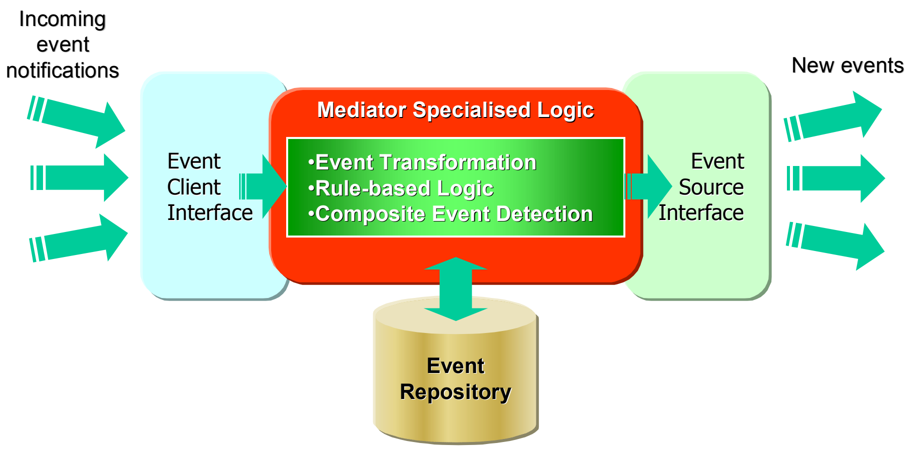

**Figure 4.10**
Structure of storage-enabled mediators

events. Otherwise, it can serve a number of remote applications. These can either interface with it through some proprietary interface or through an event source interface. See Section 7.2 for more information on this topic.

### 4.3.4 Additional applications within message-oriented-middleware

Although this discussion has focused on providing an event-based infrastructure that embeds event storage, an event repository could also be embedded within a MOM product.

Most benefit can be derived from embedding an event repository within a Message Broker that provides publish/subscribe and message transformation services. Subscribers can then ask to retrieve past information, and could carry out queries on messages generated up to some threshold in the past. The Broker itself

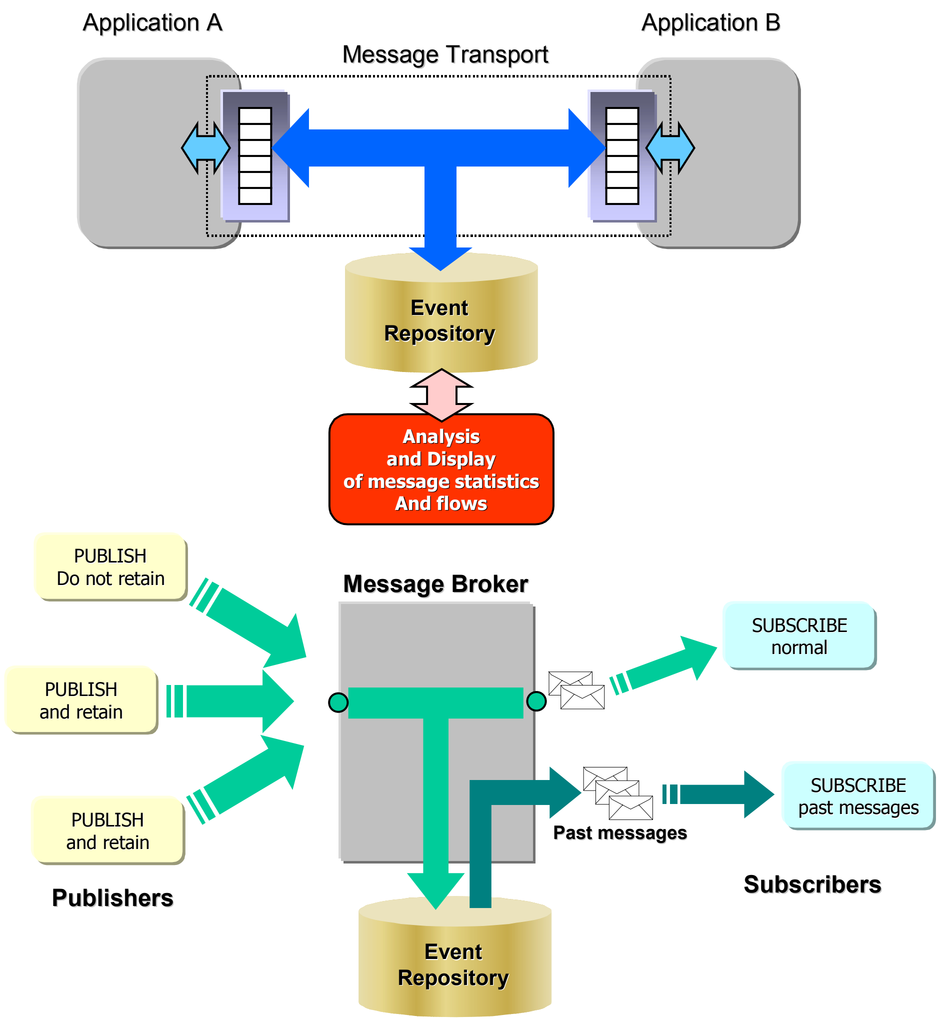

**Figure 4.11**
Embedding an event repository within MOM

could use the history at its disposal to generate message digests, which can themselves become a service that clients could subscribe to. Periodically, the broker would analyse the messages generated within a recent window of time and produce a meaningful summary. This could then be made into a message itself and propagated to interested subscribers. Since these messages could be retained themselves, future subscribers could request copies of past ones that they are interested in.

By attaching an event repository to the MOM transport itself, the transport could retain a history of the messages that are propagated by it. If the MOM is not structured around a centralised message delivery service, event storage could be embedded at the MOM modules at the sending and receiving sides.

These two scenarios are illustrated in Figure 4.11.

A MOM product can use an event repository to store events that reflect meta-data on its activities. Specifically, it can store events that track the flow of transactions within the message transport.

## 4.4 Summary

HERALD is a general-purpose event-based infrastructure that embeds event storage and retrieval. It endorses an event model where events are of a defined type and have typed parameters. Event types can belong to type inheritance trees, and their schema are versioned in order to support schema evolution. HERALD assumes deployment in a dynamic environment where components can come and go, and can be modified independently of each other.

Event storage can be embedded at various locations within HERALD modules. It supports retrieval and replay of past events, regeneration of application state, support for mobile and disconnected clients, as well as specific services within specialised mediator components.

The main components of the infrastructure presented were built and deployed in a number of application scenarios. Some of these are illustrated in Chapter 7.

**Chapter 5**

# Querying Event Information

As discussed in Chapter 2, a generic event-storage service is applicable within various application domains. The event information *deposited* into an event repository needs to be accessed, reviewed, and retrieved in different ways. These are specific to the application, and depend on the activities represented by its event information. This chapter discusses the interfacing capability required in order to address these diverse requirements.

In order to achieve this, the event repository proposed supports an interface that supports: (1) customisation, configuration and specific event operations, and (2) the execution of queries in an event query language called TEQL that explicitly supports the temporal nature of events and defines a temporal formalism. This passing of a TEQL query though a programming interface is analogous to Microsoft's **ODBC** [Mic99d] and Sun's **JDBC™** [Sun99b] database interfaces. Since several important research issues are addressed in the language proposed, this discussion focuses on the query language, while Chapter 6 describes the remainder of the interface.

Section 5.1 introduces the temporal model endorsed by this discussion and reviews the temporal entities recognised and their relationships to one another.

Section 5.2 then identifies the querying requirements of an event query language by examining the peculiar nature of event data and suggesting typical queries that one needs to address. The properties and temporal characteristics of the language are discussed.

Since there have been numerous research initiatives to develop languages that emphasise the notion of time, Section 5.3 surveys the main contributions from a number of research domains that are relevant to this discussion.

Section 5.4 then presents TEQL and describes its primary constructs. This language is designed to satisfy the query requirements of an event repository, and allows the representation and treatment of qualitative temporal relations between events, composite events and temporal intervals. It can also be used for powerful and generic data access, as it is a superset of **OQL** [CB97], which is itself a superset of **SQL-2** [ITI98].

In conclusion, Section 5.5 evaluates TEQL by demonstrating its expressive power through a number of examples.

## 5.1 The temporal model

It is useful to start this chapter by defining the temporal model and entities that will be used in the remainder of the discussion.

Reviewing the definitions given in Section 4.1, an *event* is a message that denotes the occurrence of an activity of interest, and belongs to an *event type*. An event type is analogous to an object class, and it encapsulates a number of *parameters* or *attributes*. While an event instance itself usually reflects the occurrence of some activity, its attributes uniquely identify that activity by representing its parameters.

Time can be modelled and reasoned about in several ways, with most approaches differing on the definition and relationship of time points to time intervals. Allen describes the main models in [All83] and [All91]. In this document, an *event* is taken to occur at a precisely determined point in time, *a time-point*, and has no duration. Time is assumed to consist of an infinite number of linearly-ordered discrete time-points, each of infinitesimally small and negligible duration. This is in contrast to other formalisms where events can span a determined amount of time. Because of this, every two events can be either temporally ordered, or else deemed to have occurred simultaneously. The latter can be applied if the events fall within some minimum time separation from each other over which clock inaccuracy becomes a factor.

A *time interval* I is a set of time-points between a starting and an ending time-point, where the start-point and all points between the start and the end belong to I. It is realised that this can be counterintuitive. For example, consider the situation where a person moves from a room to another. To describe a change of location one can have an interval of time during which the person was in the first room, followed by an interval where they have moved to another room. The question arises as to whether these intervals are open or closed; that is what is the nature of the time point between the two. If the intervals are open, the state at this point would be undefined, while if they were closed, it would belong to both. A pragmatic although artificial compromise is to adopt a convention where an interval is closed in its lower end and open on its upper end.

A *primitive event* is an event that is generated at some event source in response to some device's activity, and cannot be broken down into any finer-granularity constituent activity.

A *composite event* is an event that is generated to denote the presence of a particular sequence of events. It is defined as having occurred when the whole event sequence that defines it occurs, this being equivalent to the timestamp of the event that concludes the sequence. Recall that composite events can be either *regular* or *compound composite*. A composite event still occurs at a particular instant and has no duration, however, a *compound composite* event can be transformed into a *time interval* to apply queries over the duration of the sequence that it matches.

*Primitive* and *composite events* can be related between themselves or to each other, and both can be related to fully defined, or vaguely specified, *time intervals*. The latter can be user-defined or be calendar definitions.

| **Relation** | **Pictorial Example** |
|---|---|
| A *before* B | `AAA BBB` |
| A *equal* B | `AAA`<br>`BBB` |
| A *meets* B | `AAABBB` |
| A *overlaps* B | `AAA`<br>` BBB` |
| A *during* B | ` AAA`<br>`BBBBBB` |
| A *starts* B | `AAA`<br>`BBBBB` |
| A *finishes* B | `  AAA`<br>`BBBBB` |

**Table 5.1**
The relationships possible between ordered intervals, less their inverses.

An event A can occur *before* an event B, whereupon B is said to be *after* A or to *follow* A. Both terms, *after* and *follow*, are used to distinguish different query semantics within Section 5.4. Events cannot *overlap* as they have no duration.

Because intervals have duration, relationships between them are more elaborate. There are thirteen relationships possible between an ordered pair of intervals, and Table 5.1 shows the main ones (not including the inverses).

Finally, a *timeline* is the range of timepoints over which a query or operation is considered. In the case of event histories, this can range from $-\infty$ to the present, NOW. In practice, this is restricted by the fact that the history will have a specific beginning, and NOW is more likely to be a few seconds behind the current time due to the indexing and storage requirements of the event repository's storage subsystem. In this discussion, it is assumed that there is a notion of global time in the system through reasonable clock synchronisation with **Universal Time Code** (**UTC**). Section 5.2.6 discusses the implications of this assumption. Therefore, the values of timepoints on any timeline correspond to values of *real time*. This is desirable since query operations in several application domains would be made by users in terms of real time.

## 5.2 Querying requirements

This section first recalls the application classes where the repository is applicable and then reflects on the nature of queries that one needs to be able to support. It then defines the properties and characteristics of an interface language for an event repository.

## 5.2.1 Applications

As described in Chapter 2, an event repository can be used in a large variety of application domains. It is useful to categorise these as:

- *Loosely-coupled applications employing general-purpose event-driven infrastructures*
  Event-driven applications are constructed of active components that act, or execute code, in response to being notified of relevant events. These components do not need to be aware of each other and of the general structure of the application, as it is event notifications that drive computation within the environment. Events histories can be used at event sources, event brokers and gateways, both implicitly and explicitly. Chapter 4 described the role of event storage within these environments in detail. Access to stored event information tends to be dynamic and varied, with flexibility required in defining queries over multiple types, event instances, and composite event sequences.
- *Applications that employ bespoke message-propagation mechanisms for communication*
  These are applications that adopt a bespoke messaging system in order to drive their execution. It is likely that their components are aware of the general structure of the application and are dependent on each other. Centralised control structures are more predominant. Access to stored event information tends to be statically defined and of a bespoke nature within the application itself, unless the event trace is being analysed by an external application. Examples are simulation software, windowing systems, collaborative environments, and usability capture software.

Some applications demand straightforward conventional access to persistent event information, as they would with a conventional relational or object-oriented database system. In this context, the primary benefit to them of using an event repository lies in the high-speed event deposit performance provided by the store.

Others can benefit by being able to replay or retrieve stored event sequences (and have these injected back into their components), in order to review activity, analyse past computation, or rebuild lost state. The more powerful the granularity with which they can define and construct the sequences to be replayed, the greater the advantage they can extract from using an event repository service over a conventional database.

## 5.2.2 The nature of queries on event histories

The dynamic nature of execution within asynchronous event-driven environments implies a lack of predictability in the stored event data. It becomes harder (and is very limiting) to only be able to qualify the stored event information through relations on its content (as in languages like **SQL** or **OQL**). Event information differs from other data in having a temporal aspect. This tagging of an event with a *creation* or *deposit timestamp*, as well as its occurrence in time relative to other events or temporal entities, allows a completely different type of query reasoning to be applied to event data. An event can be qualified not only by its content but also based on when it occurred, and how that temporal position relates to other events' or to a calendar.

While an event repository should be accessible like any conventional database for typical queries like:

Ex.1 "find the event of type ServerCrash where the server was the ActiveBadgeServer."

it also ought to be possible to search on the basis of the temporal sequencing of the events. The following natural language sentences indicate the wide scope of temporal expression that one can come across in queries relating to event occurrences:

Ex.2 "find all occurrences of Alarm A being followed by Alarm B, and the web server crashing, within 5 minutes, and without Alarm C having occurred in between"

Ex.3 "Replay all sightings of John from when John was present in the Meeting Room, and then sometime within 30 minutes John was seen in the corridor with Giles, until …"

Ex.4 "Locate the text that was displayed on the whiteboard by Jean during the collaborative session on Monday, given that it occurred while a video on Presentation Skills was being played."

Ex.5 "Replay all the display events (mouse movements and clicks, key-presses) which occurred within the interval when both Netscape Navigator and Microsoft Internet Explorer were being used concurrently, and the latter was displaying Tim's homepage."

Example 2 involves a search for a sequence (or pattern) of events, where these would be alarm and warning events probably generated by one or more fault monitoring components in a distributed system.

Example 3 illustrates the problem that one can encounter when trying to map real-world activities to computer activities. It is possible to define this query solely in terms of primitive events, but such a definition would be complex and unwieldy. Being able to define temporal intervals, and then reason with regards to them, makes it much easier to write. One thus first wants to locate the interval during which John was in the Meeting Room. An easy way to do this with **Active Badge** location technology (although not entirely accurate), is to look for the first sighting of John in the Meeting Room (the start of the interval), and then denote the first subsequent sighting of John outside that room as the end of the interval. One then wants to make sure that this interval is the correct one by searching for John being seen in the corridor with Giles within 30 minutes from any part of the first interval. This second occurrence can be defined as the start of the intersection of the intervals during when John and Giles were in the Corridor respectively. Alternatively it can simply be defined as the sequence of a "John in corridor" sighting followed by a "Giles in corridor" sighting without John being seen anywhere else in between.

Example 4 is similar to the previous example, although it illustrates the use of events relating to drawing and presentation activities within a collaborative shared-space environment. The playing of the video can be mapped to an interval. There is also a relation to a pre-defined calendar entity (e.g. 'Monday').

Likewise, Example 5 deals with an application-based scenario, and represents a query typical of someone working with a usability monitoring tool or a thin-client session recording utility as described in Section 7.4. Three intervals are being related to each other: the running timeframes of the two web browsers and the interval during which the homepage was displayed, from which a new interval is defined. This is then used to initiate a replay.

### 5.2.3 Language properties

No repository query language can match the above natural language sentences for expressive power. However, the above expressions serve to demonstrate that a language that allows interfacing with stored event data within an event repository must address the following issues:

- *Flexibility* – It must be possible to define queries rapidly and in a flexible fashion, e.g. interactively as well as through embedding within application components.
- *Intuitive* – One should use language keywords and operators that offer the right compromise between the clarity of natural language constructs without being excessively verbose. Words with an already accepted meaning, e.g. an informal meaning, should not be given an additional meaning.
- *Conventional constructs* – It is desirable to support conventional querying capabilities as available in standard database query languages (like **SQL** and **OQL**). If excessive specialisation comes at the expense of generic data access, the language is unlikely to be suitable as the interface to a general-purpose store.
- *Type-independence* – Most database query languages allow retrieval of data of a single type in any query operation. Support for operations that span several event types in any one query is desirable.
- *Composition of events and derivation of new views* – As defined within the temporal model, one needs to allow composition of multiple-type event instances into composite events. These could be saved to generate a new derived event data-set that reflects a coarser-granularity view of the original event data. Query operators that range over both primitive and composite events are required.
- *Temporal context* – Type-independent specification of temporal relations between event instances should be supported, and a consistent formalism for time and temporal entities (calendar, or otherwise) must be available.
- *Standard type system* – New and novel query languages require time, investment, and dedication on behalf of organisations in order to train personnel. Software modification is expensive and undesirable. Due to these pragmatic constraints, a language that is close to an industry standard and supports a standard data model (like the OMG or the ODMG's data models) is more desirable.

### 5.2.4 Main temporal characteristics

While Section 5.1 has given the temporal formalism underlying this discussion, the examples in sub-section 5.2.2 have hinted at the temporal queries that may be carried out on event instances. With this information, one can now outline the main temporal characteristics required of an event query language:

*Temporal relativity between events* – Relative comparisons between event instances based on the events' temporal position along a common timeline should be supported, e.g. "A occurs before B", or "B was deposited before A".

*Temporal relativity between events and calendar entities* – It is useful to support relative comparisons based on an event's temporal position to user-defined or system defined calendar entities along a common timeline. E.g. "A occurs on Monday" and "B occurs during term-time".

*Imprecise Sequencing* – The representation should allow significant imprecision. Most temporal knowledge is strictly relative (e.g. "A is before B") and has little relation to precise time values or dates.

*Event composition* – By identifying a pattern of events that are temporally related, one can define a composite event, which is taken to have occurred when the last element making up the pattern occured.

*Variable temporal granularity* – The representation should allow one to vary the grain of temporal specification, i.e. when relating events to *real time*, one may need to consider their occurrence with regards to (e.g.) hours, minutes, seconds, or less.

*Intervals* – Elaborate temporal comparisons are not possible without a definition of temporal intervals. The representation should define an interval with respect to the temporal model adopted, and support relating of intervals to themselves, to primitive and composite events, and to calendar entities.

It is important to note that throughout this dissertation events have been considered as a "point in time" occurrence, where each event instance is inherently different from any earlier or subsequent instance of the same type. This is an important distinction as it implies that there is no implicit notion of a state for an event. This would otherwise evolve over time as event notifications of the same type occur. It is possible to model the notion of state evolution but in this context this ought to be specific to applications. This should therefore be supported at an application-level, and it is not beneficial that there be implicit support for it in the query interface.

### 5.2.5 Event sessions

At this point it is useful to introduce the notion of *event sessions*.

An event session provides a context with which to associate related event instances, and is particularly useful for limiting the scope of retrieval and replay. This grouping of related event instances can consist of events of the same or different types from one or more event sources. Any type of event can be associated with, or *belong* to, a session, and the same instance can belong to multiple sessions.

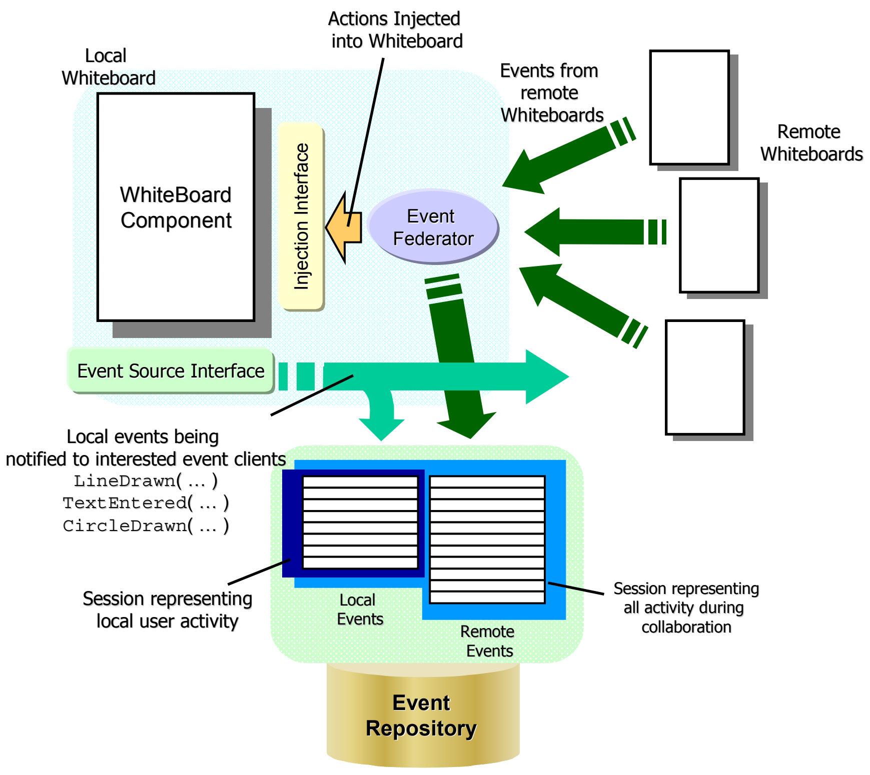
Actions Injected into Whiteboard
Local Whiteboard
Events from remote Whiteboards
Remote Whiteboards
WhiteBoard Component
Injection Interface
Event Federator
Event Source Interface
Local events being notified to interested event clients
LineDrawn( ... )
TextEntered( ... )
CircleDrawn( ... )
Session representing local user activity
Local Events
Remote Events
Session representing all activity during collaboration
Event Repository

**Figure 5.1**
An example of deposited events being tagged as belonging to either one or two sessions.

A session is set up by an application (or that application's representative *repository client*, as described in Chapter 6). A session therefore enables an application to define a context or relation between events it deposits in a repository, so that it can then retrieve them at a later stage based on that context.

An example is a person's diary containing all events pertinent to their activities, or a chronicle of every individual's movements while exiting the building in an emergency drill. Another example would be associating the line drawing activities of an individual user within a whiteboard application through a session, as illustrated by Figure 5.1. A replay on the session context would then enable only those events to be replayed back and injected into the whiteboard. However, at the same time, those events could also be associated with another session that also encompasses events relating to drawing activity by other remote users.

## 5.2.6 Event timestamps

HERALD event instances deposited into the event repository have two timestamps:

- *Creation timestamp*
- *Deposit timestamp*

The *creation timestamp* is applied at the source where the event was created, while the *deposit timestamp* is attached automatically to an event instance while it is being queued for deposit into the event repository. When the event repository is attached to an application that is generating the events that are being stored, or when it is embedded within an event source, there is no need to distinguish between the two timestamps. In fact, in this case the creation timestamp can be set to a null value.

When, however, the event repository resides in an application component that is a remote observer of events generated elsewhere, the situation is different. In an ideal world, these values would be identical, or at least the difference between them would be constant. In practice, however, several factors can contribute to prevent this from being so. As discussed in Section 4.2.4, in a distributed system there is no built-in notion of a global clock, and each machine has its own clock that may be out of synchronisation with the other machines' clocks, and drifts unpredictably. Furthermore, once an event is generated and timestamped, the event is queued at that source, dispatched, queued again at the recipient, and finally passed into the repository for storage. The time spent in being transferred over the network is not constant as the network latency varies according to the network load and, in case of an IP-based network, the route taken.

One can ensure that events from a particular source are correctly ordered with respect to each other at the event repository's location by providing a sliding time window for observation, into which incoming events are insertion-sorted. This can be achieved either by relying on their creation timestamp or on a scalar counter attached to them by their source.

The same cannot be guaranteed of events from different sources, since their creation timestamps are insufficient for ordering due to their having been produced at potentially unsynchronised clocks. In a distributed application where processes communicate using messages, *vector* [Fid91, Mat88] or *matrix* [FM82] *logical clocks* could be used to enforce global ordering of messages. In a general-purpose event-based system, however, these are not easily applicable because communication is mostly asymmetric and the number of interacting components is variable.

What is definite is that the *deposit timestamp* reflects the order and temporal separation of the events as received, or else accepted, by the client or observer where the events are to be stored. This therefore reflects the viewpoint of the application interaction as seen by the event observer. In most cases, **NTP** [Mil91] coupled with modern computing hardware, provides for clocks to be within 10*ms* of each other and of **UTC** anyway. This, in conjunction with a sliding time window of application-specific length for event observation in order to counteract network delays, is enough for most observations. For this reason, all query operations within TEQL are carried out with respect to the deposit timestamp and its implied ordering.

If the observer is concerned about guaranteed 'global-time' event ordering, and **NTP** is too imprecise to enable ordering the events being received (maybe because too many events are within 20*ms* of each other [Kop92]), two approaches are possible:

- the application can provide a virtual time mechanism [Mat88], on top of the event propagation protocol,
- rather than locate an event repository at a centralised location where it acts as a remote observer of events, multiple event repositories can be located in close proximity to the event sources, and their traces subsequently merged through the repository federation facility described in Section 6.5.

This second approach is recommended by Dietz from her comprehensive study of gathering and using time measurements in distributed systems [Die96].

As already qualified by Section 4.2.4 for HERALD, all timestamps in the event repository consist of two parts. One is the component identifier of the event component where the timestamp was attached, while the other consists of two long integers that denote the number of microseconds since 1st January 1970. This associates the system's notion of *global time* with *real time*, which is desirable given that most user-specified queries are envisaged as being with respect to real time.

## 5.3 Related work

The literature on languages that emphasise the notion of time is rich and growing. There have been numerous research initiatives to develop and implement *temporal/historical* query languages. The main areas where such query languages have been investigated are:

- *Event languages within event-driven environments*
- *Event languages within Active Databases*
- *Temporal/Historical Databases*
- *Artificial Intelligence*
- *Natural Language Processing*

It is only recently that middleware infrastructures, such as those described in Chapter 3, have proposed and deployed general-purpose event notification services [OPSS93, GKP98, OMG97, OMG98a, SCT95]. However, most of these can notify primitive events only, and as such do not support specification of composition of events.

Researchers at Cambridge were amongst the first to propose the *publish-register-notify* paradigm [BBHM95]. The **Cambridge Event Architecture** (**CEA**) (Section 3.3) enables distributed applications to be composed of heterogeneous software components through registration in primitive or composite events, and supports mobility, multimedia, and group interaction in collaborative environments [BHB96, BBMS98]. *Mediators* provide the means to compose events. Within this context, Hayton proposed a language for expressing composite event semantics [Hay96]. **CEA** has recently been enhanced through the **COBEA** implementation [MB98], which extends the **CORBA Event Service** [OMG97] with the publish-register-notify paradigm, and provides an implementation of an evolution of Hayton's language.

Bates and Nelson have presented a composite event specification language [Nel98] that not only supports the basic *happened-before* relation in defining a temporal ordering of events but also supports multiple *consumption models* (see discussion in Section 5.4.6). An implementation of this language was deployed in the context of location- and context-aware computing. Use of this language is also demonstrated in [BBMS98], where it is employed to trigger actions within *event federators* (see Section 4.2.5) according to the **CEA** paradigm. This is in analogy to *Event-Condition-Action* rule use within *active databases*, but applied to a distributed dynamic general-purpose context. As documented in Chapter 4, HERALD is an evolution on the framework presented in [BBMS98] and encompasses the Bates composite event engine for detecting and manipulating composite events. In addition, HERALD enhances the core **CEA** paradigm by providing persistence related registration policies, possible due to its source components embedding an event repository.

**EVE** [TGD97, GT98] models a *workflow enactment* system as an event-based middleware layer. The workflow is mapped to services and brokers. Distributed events are composed into composite events and drive the actions to be taken by event brokers. **EVE** uses the *chronicle* consumption model (or *parameter context*) to interpret workflow notifications. **Yeast** [KS95] provides temporal connectives in order to enable monitoring of compound event patterns in the context of server monitoring. Other notable examples of event composition languages are **GEM** [MS97], which evolved into a general-purpose event composition language from a distributed systems monitoring background. Trinity College Dublin's **ECO** model [SCT95] does not support composite events, but it enables post- and pre- constraints to be attached to the notification of events.

Composition of events was first investigated within *active database* systems [DBM98, GJS92, CM93, WC96]. This took place within the context of *Event-Condition-Action* (ECA) rules. These rules enable a database action to be triggered by the occurrence of an event or a specified sequence of events. The events supported and monitored for internally by an active database can be: (1) events that denote specific database activity, 2) timing/temporal events, and (3) user-defined events, which usually reflect some change in state of the database's data. From the work illustrating architectures for composite event recognition within active databases, representative examples are [GJS92, GD93, CKAK94]. **HiPAC** [Day88] focused on general-purpose ECA rules and provided basic mechanisms for composite event specification. Four consumption models and a formal definition of event composition were introduced within **Snoop** [CKAK94]. **Compose** [GJS92] and **Ode** [JS94] both provide complex event composition.

Some researchers have attempted to extract this functionality from within the internal architectures of active databases and make it available in a more general-purpose form [GKBF98, KK97]. This is analogous to what event-driven systems like **CEA** have achieved by addressing distributed systems composition and interaction directly.

Most of the languages discussed above are aimed at real-time detection of events, and are thus not optimised for operating on a full event history such as that provided by an event repository. Queries cannot be constrained by periods of time or calendar entities, and cannot be applied to event streams that have occurred in the past. In addition, there is no support for expressions based on the end-points of composite events, and on notions of temporal interval relations, as proposed in Allen [All83]. These are required for expressiveness in real-world complex queries.

In *temporal* and *historical databases*, whenever a data element is updated, its previous value is retained, and the state of the element evolves. Looking back in time though the database, it is possible to review and reason on the value of any element at any time. Temporal and historical database query languages support only very limited forms of temporal reasoning, as the emphasis in these languages is to enable reasoning based on previous values of the state of a data element, and on determining the overlap in time between states of different elements. There is thus no support for the notion of sequencing between different data elements. Nevertheless, aspects of these languages are of relevance to this research. Some of these languages, in particular **SQL**-based approaches like **TSQL2** [Sno95], support Allen's like qualitative relations in queries [Sno93, TCG+93].

Of interest is Jagadish et al. [JMS95]'s **Chronicle Data Model**. This model addresses the requirement of transaction recording systems to store transaction records in temporal sequences. A **Summarised Chronicle Algebra** (**SCA**) is presented that can be used to define persistent views over the sequences.

Approaches in *artificial intelligence* (AI) and *natural language processing* focus on high-level user-friendly formalisms for representing and reasoning on time. Many AI treatments pay most attention to the treatment of temporal quantifiers (like *once* and *every*) and e-Time (*during* some activity, etc). Examples are [Lad86a, Lad86b, Lig91, MKL95]. The aim behind these treatments is to develop a form of temporal reasoning for checking the consistency of a knowledge base of temporal facts, and for deriving new temporal constraints from it. Following Allen's proposals [All83], most of these provide algebraic approaches in which a specialised formalism is introduced to represent temporal information. These languages tend to be very expressive and are closest to natural language expressions. However, they are ill suited for application in information retrieval and querying because the high-level nature of their constructs is not easily mapped to fine-granularity information sources like events. Nevertheless, an exciting future research area will consist of investigating the mapping of these AI-derived languages onto a query language such as that proposed in the next section.

## 5.4 The TEQL language

This section proposes the basic elements of a language called TEQL that addresses the requirements identified earlier. TEQL encapsulates **OQL** [CB97] within it, but is more than just a superset of **OQL**. TEQL allows queries on content to be defined in terms of **OQL** constructs like the select-from-where statement. However, such reasoning is unwieldy when it comes to comparing event instances to one another, like with respect to those events that precede or follow them. In order to address this lack of ease in specification within **OQL**, TEQL allows specification of *content templates*.

This section emphasises the constructs and operators of TEQL rather than **OQL**. As the data model within TEQL is compatible with **OQL**'s, **OQL** constructs and the new TEQL constructs can be seamlessly used together. For a complete definition and discussion of **OQL**, the reader is referred to Chapter 4 of [CB97]. **OQL** was chosen as the basis of a query interface to the event repository because:

- it is a powerful database query language that can satisfy "conventional" access requirements relating to content queries:
- it is an industry-defined standard for interfacing to object-oriented databases, and its object-oriented extensions can be applied to the inheritance-based event model presented in Section 4.1,
- it is largely a superset of **SQL-2** (or **SQL-92**), which is the most widely deployed database query language,
- it simplifies interfacing legacy and conventional non-messaging applications to the event repository,
- there is extensive experience and investment by companies in **OQL** and **SQL**, and this consideration needs to be applied against the case for defining a completely new query language.

The only important feature of **OQL** that is not applicable here is object method invocation. This is because the event 'objects' stored do not define any methods.

The temporal functionality of TEQL takes as its basis the query capabilities described in [Nel98]. This composite event language is restricted in its power because it does not have access to a complete history and can only carry out real-time monitoring. TEQL is able to take the expressiveness further because it can look forwards and backwards across a complete timeline.

### 5.4.1 Creating events

There are two ways of depositing events into the event repository. Both require the scheme of the event type to have been defined in advance.

In the first case, an event stream for each type can be opened to the repository, and event instances passed in (as they are received) through the repository server's command interface (see description in Chapter 6).

The alternative is to explicitly specify event instances for depositing as follows:

> deposit <*event type*> (<*creation timestamp*>, <*source component id*>,
> [<*parameters*>]);

All HERALD event instances encapsulate a *creation timestamp* and a *source component identifier*. These refer to when and where the event instance was created, respectively. If the event is being created by the component at the interface itself, then the first two parameters may be omitted. In any case, the primary *deposit timestamp* of an event is implicitly set by the event repository upon insertion of the event instance. An example of this would be:

```
deposit PersonMovement("Mark", "Room4");                                (1)
```

and if the event instance is also being attached to one or more sessions, like for example, to two sessions called MarkDiary and PersonnelLocation:

```
deposit inSession MarkDiary, PersonnelLocation PersonMovement("Mark",
    "Room4");                                                           (2)
```

The unqualified event type name PersonMovement can be used only if it is clear in the current context, i.e. if there is only one definition for it in both the inheritance and evolution trees (see Section 4.1). If the former case applies, then it can be specified as:

```
deposit Event.LocationEvent.GPS_Sighting (...)                          (3)
```

while in the latter case, the following is required:

```
deposit Event.LocationEvent$<component><version>
    .GPS_Sighting$<component><version> (...)                            (4)
```

This fully qualifies the scheme of an event type by specifying the components that defined it, and the version of its definition within the context of that component. The <*component*> and <*version*> identifiers can be obtained through the repository's API.

## 5.4.2 Temporal primitives

The following three sub-sections introduce the four temporal entities supported by TEQL. They are *primitive events*, *composite events*, *intervals*, and *timepoints*.

### Event templates

Whereas **OQL** relies on specifications such as:

```
select A
    from PersonMovement A
    where A.personName = "Mark"                                         (5)
```

in order to locate events of the type PersonMovement("Mark", ?), TEQL also allows the specification of event templates (as in Nel98). Templates are more straightforward to define, are analogous to registration templates (as in HERALD), and enable multiple compositions to be drawn up more easily. This will be demonstrated in the following sections. A template specification equivalent to the above would be:

```
match PersonMovement("Mark", ?) as A;
retrieve A;                                                             (6)
```

where the first parameter's value must be equal to the literal "Mark", the second parameter can have any value, and the variable A is used to denote the resulting event template. As used in example (6), the retrieve clause searches for instances matching the template over all the repository data. More information on this clause is given in

Section 5.4.7. Each parameter needs to be either a literal value, an expression or function that computes to a literal, or the symbol ?.

The power of template matching starts to become evident when temporal sequencing is introduced into a query. One can deduce when a computer terminal has been logged out from by its user for the evening by:

```
match TerminalLogout("Mark", "Room4", ?) as A;
match PersonMovement ("Mark", "Lift") as B;
match (A before B within 15min) as ...                              (7)
```

where in this context the qualifier before is syntactically analogous to *which occurs before*. The final template will match all instances of A that occur before an instance of a B within five minutes. If variables are not used, the above can be written as:

```
match TerminalLogout("Mark", "Room4", ?) before
    PersonMovement ("Mark", "Lift") within 15min ...                (8)
```

match always denotes a template of the same type as its first parameter. While in (7) and (8) these were event templates, they could also be composite event templates or interval templates (see below).

## Composite Event templates

In order to specify a composite pattern of events and implicitly generate a new composite event, one can specify a composite event template in terms of other event templates as follows:

```
match (A follow B past 5min) as C;
retrieve C;                                                         (9)
```

where A and B are event or interval templates. The retrieve clause will now return an ordered set of composite events, with each event having occurred when the pattern denoting it finally occurred. In the above example, for each composite event returned, this would correspond to the time at when the B within it occurred.

In this case, however, these composite events are virtual structures that are untyped and do not have user-defined parameters. What they encapsulate is an internal list that points to the events that make up the pattern they represent. In order to create new compound composite typed events that can be retrieved and propagated through the event infrastructure a scheme must be defined within the query as follows:

```
match (A follow B past 5min) into NewEvent(A.parameter2, B.parameter1)
    as C;
retrieve C;                                                         (10)
```

This actually types the compound composite events matched as being of type NewEvent having two parameters, whose types are automatically extracted from the values passed. In this case, both type and value are taken from the parameters of the matched A and B events. It is important to write the query such that A and B's type can be determined.

Therefore, whereas A before B is analogous to "*the A that occurs before a B*" and is a qualifier on its first parameter; follow is an operator which returns "*the sequence of A followed by B*".

The primitive events making up the sequence within a composite event can be retrieved through

```
componentOf(C, n)
```

which returns the n'th plus one event that makes up the composite event C. As in **OQL**, lists and set elements are counted starting from 0.

## Intervals

The final temporal construct in TEQL's temporal formalism is the interval. As described in Section 5.1, an interval is defined as a period of time bounded by a start and an end time-point. The start-point is taken to belong to the interval while the endpoint does not. An interval can be created as follows:

```
interval A, B
```

where A and B denote time values. If A and B are event (primitive or composite) templates, then their storage timestamps are extracted and used to denote the interval endpoints. A is said to *belong* to the interval, while B *meets* it. The only assumption on the relationship of A and B made by the interval clause is that B must always occur after A, and which A and B are selected depend on the parameter context (see Section 5.4.6). Other examples of interval definition are:

```
interval 08/01/1975-22:10:30.15, 10:30 as I;                  (11)
interval PersonEnters, PersonLeaves as Stay;                  (12)
interval 15:00, 30min as Siesta;
retrieve Siesta;                                              (13)
```

Example (11) illustrates the simultaneous use of an *absolute timepoint* and a *query-time-related timepoint* (see next sub-section). Example (12) illustrates how two event templates can be used to define an interval.

(13) then demonstrates a different way of defining an interval; by its length. This mode is invoked by supplying a quantitative time value as the second parameter, e.g. 30 minutes. In example (13), the interval Siesta is defined as being the time from 3pm to 3:30pm. If the second parameter had been negative, it would have meant 2:30pm – 3:00pm. The retrieve clause returns a set of intervals matching the *interval template* Siesta.

An interval's boundary point can be denoted by another interval.

```
interval A, Siesta                                            (14)
```

denotes an interval from the event A to the beginning of the interval Siesta, while

```
interval Siesta, A                                            (15)
```

denotes the interval from the end of the interval Siesta to the event A. Interval templates may be used within match constructs in a way similar to primitive and composite event templates:

```
match Siesta before PersonEnters within 15min as Return;                    (16)
match Stay follow Siesta within 1hr as AfternoonLull;                       (17)
```

(16) defines the interval template Return, which consists of all Siesta intervals that are followed by a PersonEnters event within 15 minutes. In (17) a template denoting the occurrence of a new composite event AfternoonLull is defined. This consists of the sequence of an interval matching Stay being followed by an interval matching Siesta within an hour.

### Timepoints

As a language that emphasises the concept of time, TEQL supports several ways of specifying time in queries.

A fully qualified or *absolute* timepoint is defined as

*<year>*/*<month>*/*<day>*-*<hours>*:*<minutes>*:*<seconds>*

For example:

```
1998/08/01-22:10:30.15
```

The qualification moves from left to right over the definition, with only the first value required. Therefore 1975/08 is acceptable and is syntactically equivalent to 1975/08/01-00:00:00. Inversely a time point can be defined from right to left, in which case it is called *query-time-related* timepoint and is evaluated with reference to the time when the query is executed. Therefore:

```
15:30
01/15-20:30
```

stand for 3:30pm on the current day, and 8:30pm on the 15th of last January, respectively. Resolving a definition of this kind may fail and cause a run-time error if the query is run before the time represented.

## 5.4.3 Amounts of time and calendar entities

### Timepoint-denoted intervals

An absolute time value may be used to denote an interval by preceding it with '*'. The interval will be as long as the highest defined granularity within the time value. For example:

```
*15:30
*1999/01
```

represent from 15:30 to 15:31, and all of January 1999.

### Time amounts

Amounts of time can be specified in any number of seconds (sec), minutes (min), hours (hr), days (dy), weeks (wk), months (mth), and years (yr) by attaching the respective qualifier to the quantity. Examples are

```
30sec, 5.6min, 10yr
```

### Calendar entities

Calendar entities are like pre-defined interval templates relating to calendar periods. There are two predefined sets of calendar entities, the days of the week, and the months.

```
match A during Monday
```

defines a template that matches on those A's that occur on any Monday within the period of time during which the query is applicable (*the current temporal scope*).

TEQL could be further enhanced to allow users to define their own calendar with its calendar entities (*I-Times*). This would require a formalism for specifying I-Times such as that proposed by [Ter97].

## 5.4.4 Basic operators

The basic temporal operators on the temporal entities are now more fully described.

### Qualification, occurs-before

The full definition of a before qualifier is

```
A before B [without C] [past T1] [within T2]
```

This matches "the A that occurs before a B within T2 amount of time but there is at least T1 time between A and B, and no C occurs between A and B". All A's that match the above would be matched.

past introduces a minimum amount of time that must pass between the two events, while within restricts the maximum amount of time that can pass. In the above example A and B can either be primitive events, composite events, intervals, or timepoints. If both operands of a before are interval templates, the endpoint of the first interval has to be larger than the starting point of the second interval. If a within clause is not defined, the definition is taken to be *open*, i.e. valid until the end of the current timeline. The temporal entities matched are always of the same type as the type denoted by the A template, that is one of a primitive event, composite event, interval or a timepoint.

```
A before B within T                                    (18)
(A before B before C) within T                         (19)
(A before B within S) before C within T                (20)
A before B without C within X                          (21)
```

(A before B before C) without C within X (22)

The scope of the optional within and without qualifiers covers the last before qualifier, unless braces extend their range over multiple before qualifiers, as in examples (19) and (22). Example (19) illustrates nested use of before, where one wants to match an A that occurs before a B, but the B occurred before a C, and no more than T time must have elapsed between A and C.

## Qualification, occurs-after

The "occurs after" qualifier after is semantically the inverse of the before qualifier.

A after B [without C] [past T1] [within T2]

The only difference from the before qualifier is that this matches "the A that is preceded by a B in T amount of time, and no Z occurs between A and B".

## Composite sequence, followed-by

The full definition of a follow clause is

A follow B [without C] [past T1] [within T2]

This means "the sequence of an A that is followed by a B within T2 amount of time, no C occurs between A and B, there is at least T1 time between A and B".

follow is an operator rather than a qualifier, in that it returns a new composite event template, with the timestamp of the composite event being the time when the whole sequence was matched (or occurred).

Otherwise, the same discussion applies to its qualifiers without, past, and within as for the before qualifier.

## Disjunction and conjunction

These operators are defined as in [Nel98]. Disjunction (or) is notionally *exclusive or*.

match (A or B) follow C as Y (23)

means that the composite event template Y will match the sequence of either A followed by C, or B followed by C. Care must be taken when or is employed in entity qualification, as in:

match (A or B) before C as Y (24)

as this will match the A that occurred before C, or the B that occurred before C, where A and B could be templates denoting different event types, or even different temporal entities.

On the otherhand conjunction is available through the and operator and may only be used alongside sequence constructs. In:

match (A and B) follow C as Y (25)

Y will match when both events A and B have occurred but the order is unspecified. (A and B) is analogous to (A follow B or B follow A). and can be qualified with within and without clauses.

## Interval operators

TEQL defines a number of additional operators specific to intervals. Of the thirteen possible relations between temporal intervals defined by Allen [All83] (see Section 5.1), seven are explicitly supported by TEQL.

In addition to before and after, there are also the following qualifiers:

A isIn I – A occurs during interval I

A isOut I – A occurs outside interval I

I1 intersects I2 – interval I1 intersects with interval I2

I1 contains I2 – interval I1 contains interval I2

I1 notContain I2 – interval I1 does not contain interval I2

where A can be either a primitive event, a composite event, a timepoint, or an interval-template itself.

An interval can also be defined as the:

```
intersectionOf(I1, I2)
```

which returns a template denoting the interval defined by the intersection of intervals I1 and I2. This operation may return NULL.

The events occurring within an interval can be accessed through the function:

```
elementOf(I, n)
```

This returns the n'th plus one event that occurs within the interval I in the current scope, or through:

```
elements(I)
```

which returns an ordered (by time) set of events. On the other hand, start(I) and end(I) return the endpoints of the interval.

## 5.4.5 Variable matching

All the earlier examples have illustrated how variables can be used within a query to denote a template. However, variables can also be used within template specifications to tie together parameter values. One can re-write (8) as follows:

```
match TerminalLogout(User, "Room4", ?) before
    PersonMovement (User, "Lift") within 15min …                    (26)
```

which will identify all conclusive terminal logouts in Room 4 (assuming one sole user per terminal, and that users take the lift on their way out of the building!). Any term that is not a keyword and not a literal is interpreted as a variable, whose type is defined by its first appearance. Subsequent uses need to be on parameters of the same type. A literal can be used to initialise a variable, which can be useful in comparing values across events:

```
match TerminalLogout("Mark" as User, Room, ?) follow
    TerminalLogin(!User, Room, ?) ...                                    (27)
```

This stipulates that the first parameter of TerminalLogin must not be the same as the first one of TerminalLogout, i.e. not "Mark", while the second parameter, Room, is unspecified but must be identical over both events. In addition to this implicit request for equality, and the explicit non-equality denoted by ! preceding the second use of the variable, other qualifiers are >, <, >=, <=. These may be used on those types that support comparative relations.

## 5.4.6 Parameter contexts

Any discussion of composition of events needs to consider *parameter contexts*, also known as *consumption models*. The concept of parameter contexts was first described in [CM93] and then refined in [CKAK94]. The best way to illustrate the concepts behind the term is through an example. Consider:

```
match A follow B ...                                                     (28)
```

where A and B are templates. Now consider that the query evaluator is going to look for this sequence within the event stream:

```
A1, A2, B1, A3, B2, A4, A5, B3, B4
```

Where the A*n*'s match the template A and the B*n*'s match B. The issue at point is which sequences of A's and B's will trigger the sequence in (28). In the most general case, termed by [CKAK94] as the *unrestricted* context, and also known as the *cross-product*, the possible sequences are fifteen in all, and are:

```
(A1, B1), (A1, B2), (A1,B3), (A1,B4), (A2, B1), (A2, B2), (A2,B3), (A2,B4), (A3,
    B2), (A3,B3), (A3,B4), (A4,B3), (A4,B4), (A5, B3), (A5, B4)
```

It is expensive and often unnecessary to evaluate sequences in the unrestricted context. The context that TEQL defaults to is known as the *chronicle* context, which uses the first event seen of each class. Within this context TEQL matches:

```
(A1, B1), (A3, B2), (A4, B3)
```

for the event sequence of example (28). Although this may be suitable for the majority of retrieval applications, it is appreciated that imposition of an arbitrary context is restricting. In order to address this, TEQL, like Nelson's **CE** system [Nel98], supports two variations on the consumption model.

By preceding follow with the > modifier as follows:

```
A >follow B ...                                                          (29)
```

multiple evaluations of the expression are allowed to take place at any one time. When an A-matching entity (such as A1) is encountered, the query evaluator searches forwards for a B, but if it encounters another A in the meantime (such as A2), it launches a separate second search for a sequence starting with that entity. The resulting matches are then:

```
(A1, B1), (A2, B1), (A3, B2), (A4, B3) (A5, B3)
```

On the otherhand:

A follow< B ... (30)

causes the evaluator to first match, and then replicate and continue searching, on the second parameter. In this case the result is:

(A1, B1), (A1, B2), (A1, B3), (A1, B4)

Placing both modifiers together as in

A >follow< B ... (31)

matches on the unrestricted context. Parameter contexts are also applicable to qualification on events (as with before and after), and to interval definitions.

## 5.4.7 retrieve and replay

The main constructs introduced by TEQL over **OQL**'s are the replay and retrieve constructs.

### retrieve

The format of the retrieve construct is:

retrieve <*template_expression*> [fromSession <*session*>] [in <*interval*>]
[<*ordered set of events*>] [fromRef <*result reference*>]

The constraints define the scope over which the retrieve operation is to run. If no constraint is specified the scope of the retrieve is taken to be the entire repository, from the beginning of time (this being the first event deposited) until NOW. The constraints can be:

- fromSession <*session*> - look for matches for the template only through events belonging to the session specified.
- in <*interval*> - look for matches for the template only within the interval specified. This has to be an instantiation of an interval rather than an interval template.
- <*ordered set of events*> - look for matches for the template within the stream of events denoted by the set of events provided. This set could be the result of a nested retrieve operation, or of an **OQL** select-from-where construct on the repository.
- fromRef <*result reference*> - look for matches for the template only through events belonging to the set of events identified by the reference. A reference is returned programmatically with each result set and its associated events are cached for some time.

retrieve always returns an ordered set of temporal entities of the same type as the template expression passed to it as parameter.

Since retrieve returns sets of entities, it is worthwhile mentioning how collections, such as sets, can be manipulated in TEQL. Just as in **OQL**:

    list(a,b,c,d)[1]

returns b. And:

    list(a,b,c,d)[1:3]

returns list(b,c,d).

### replay

replay is probably the operation most identifiable with the notion of an event repository. The replay query in TEQL is very straightforward:

replay <*interval*>

replay <*ordered set of events*>

replay fromSession <*session*>

where the first variant will replay all the events which have occurred within the interval specified. The interval supplied needs to be a defined instance. In the second case, the events to be replayed will be the events within the set supplied. In both cases, it is likely that the operand would have been obtained through a retrieve query. The third variant is equivalent to the second but where the set of events is taken to be the entire session from its start to the last event deposited.

By using together replay, retrieve and select (see Section 5.4.9) it is possible to accurately specify a sub-set of events to be replayed.

Once the interval, or set of events, to be replayed are evaluated, the event repository starts sending out the stored events as they occurred, with the temporal separation reflected by the difference of their *creation* or *deposit timestamps*. This can be modified with the by clause to force faster or slower replay.

replay <*ordered set of events*> by 2

will replay the events at twice the original speed. If a fraction is used, the replay will be slower than the original deposit.

A replay can also be invoked directly through a repository-supported method. In this case, a reference to a result set must be passed as parameter of the method. This reference must qualify an ordered set of events or an interval, and can be obtained within a programming environment from earlier retrieve queries.

## 5.4.8 Derived sessions

As described in various applications in Chapter 2, it is often desirable to generate a derived view of an event history in order to assist interpretation of the data. This view frequently has to contain selected primitive events and/or composite events extracted from the original event history. In order to support this, one can create a *derived session*, which differs from a regular session in that it can be inserted into and modified.

```
create derivedSession <sessionName>
create derivedSession <sessionName> <ordered set of events>
```

allow the creation of the derived session, with the second instantiating it with the set of events provided, this usually being the result of a nested retrieve operation or of an **OQL** select-from-where construct. Composite and primitive events can then be inserted into the derived session as follows:

```
insert intoSession <sessionName> <ordered set of events>
```

A derived session is therefore analogous to a derived relation in **SQL**. It is equivalent to a regular session for all other purposes, and can be used in all constructs where a session can be used. Multiple ordered sets of events can be inserted into a derived session during different queries, thus gradually constructing a temporally ordered view that represents some high-level interpretation of the original event data. Since retrieve operations can be restricted to sessions of all types, a derived session can be used to derive another derived session of coarser granularity.

## 5.4.9 TEQL and OQL

The TEQL constructs fit in alongside **OQL** and can be used together as long as there are no type conflicts. Consider:

```
match A before B within T as X;
define theXEvents() as
    retrieve X fromSession S;
select e.parameter1
    from theXEvents() e
    where e.parameter2 = literal and e.ofType(anEventType);                (32)
```

Example (32) illustrates how this can be achieved. First one finds all events matching A that are followed by a B within time T, and belong to session S. Then from all these matching A's only those whose second parameter has a specific value, and are of the event type anEventType are selected. From this selection, the select clause then returns the first parameter, or more accurately, a set of elements of the same type as the first parameter. Of course, it would have been simpler to specify the fixed value of parameter2 within the template statement defining A.

ofType() is a pre-defined method that can be invoked on all event types to check their type. It returns a Boolean value reflecting whether the event type it is invoked within the context of is equivalent to the type name passed as parameter.

**OQL** allows objects, their data and their methods, to be qualified according to their position within the inheritance tree. Event types are mapped to the same object-like inheritance model so that such qualification can be employed within **OQL** constructs.

## 5.5 Evaluating TEQL

This section demonstrates how TEQL can be used to express queries on event information.

A straightforward way of doing this is to return to the natural language expressions used as examples in Section 5.2.2. It is often the case that there are multiple ways of expressing a query in terms of the temporal entities supported. Some may be more intuitive to write or follow, while others will be more concise. Consider first:

Ex.2 "find all occurrences of Alarm A being followed by Alarm B, and the web server crashing, all within 5 minutes, and without Alarm C having occurred in between"

This suggests searching for all occurrences of the composite event defined by the sequence of the first alarm and the second alarm. TEQL syntax for it is:

```
match (Alarm("A") follow Alarm("B") follow ServerCrash("web")) within 5min
    without Alarm("C") as theSequence;
retrieve theSequence;
```

Example 3 is more complex, and while it might not necessarily be indicative of a frequently carried out query, it is useful to consider it for demonstrating TEQL's expressive power.

Ex.3 "Replay all sightings of John from when John was present in the Meeting Room, and then sometime within 30 minutes John was seen in the corridor with Giles, until John logged in on workstation Norton."

The wording of the example can be mapped to three intervals reflecting John's stay in the Meeting Room, and both his and Giles' time in the corridor. From the latter two one needs to locate the time when both were together, and then relate that to the first by checking whether it occurred within 30 minutes. In order to define the intervals themselves one can locate the events representing the state of presence in a room. This is applicable only within the default *chronicle* context. Figure 5.2 illustrates

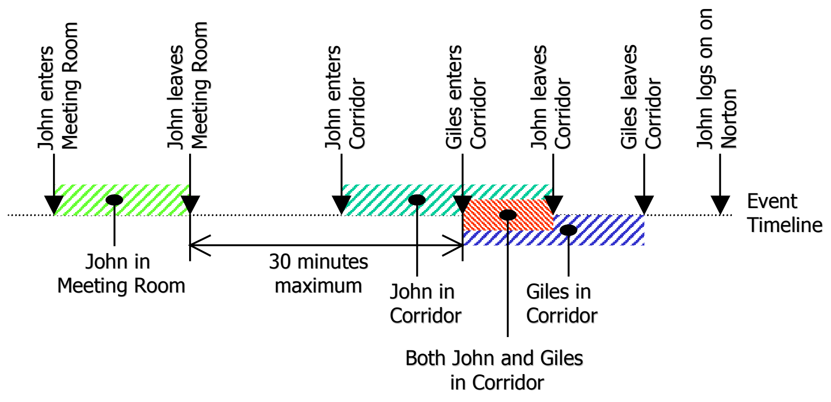

**Figure 5.2**
The events and intervals involved in Example 3

the events and intervals involved in mapping the natural language query of Example 3 to a query in TEQL.

A suitable full expressing of the example in TEQL is:

```
interval BadgeSighting("John" as John, "MeetingRoom" as MeetR),
    BadgeSighting(John, !MeetR) as iJohnMR;                                [1]
interval BadgeSighting(John, "Corridor" as Corrd), BadgeSighting(John, !Corrd) as
    iJohnCorr;                                                             [2]
interval BadgeSighting("Giles" as Giles, Corrd), BadgeSighting(Giles, !Corrd) as
    iGilesCorr;                                                            [3]
match iJohnMR before intersectionOf(iJohnCorr, iGilesCorr) within 30min as
    iJohnMR_OK;
interval start(iJohnMR_OK), TerminalLogin(John, ?, "Norton")
    as iReplayInterval;                                                    [4]
define firstMatchingInterval() as
    (retrieve iReplayInterval)[0];
replay (select events
    from elements(firstMatchingInterval()) as events
    where events.ofType(BadgeSighting) and events.user = John);            [5]
```

Parts [1], [2], [3] and [4] define templates for the primary intervals of the query.

[1] locates intervals when John was in the meeting room, [2] locates intervals when John was in the corridor, while [3] locates intervals when Giles was in the corridor. [4] then defines the interval during which both John and Giles were together in the corridor, and uses it to cut down the set of all intervals of when John was in the meeting room, to those that occurred within 30 minutes before the John/Giles encounter. A new interval is then defined from the beginning of John's time in the meeting room until the first subsequent time he logged on-to Norton. This defines suitable intervals for replay.

One should point out that [1], [2] and [3] are employing the property that:

```
interval A, B as I;
```

is equivalent to

```
match A follow B as AB;
interval componentOf(AB, 0), componentOf(AB, 1) as I;
```

While the maximum cut off time of 30 minutes was from the end of John's stay in the meeting room, the example requires replay from the beginning of that stay. It also requires replay of only **Active Badge** events, so rather than replay the satisfying interval, the query extracts the events within the interval and then filters them on type and content (see [5]). Since theoretically there may be multiple intervals that will match the above specification, only the first interval is being considered from the list of results returned by the retrieve construct.

Note that the above example could also have been evaluated solely in terms of composite sequences by replacing [1], [2], [3] and [4] with:

```
match BadgeSighting("John", "MeetingRoom") follow BadgeSighting("John",
    !"MeetingRoom") as cA;
```

```
match BadgeSighting(John, "Corridor") follow BadgeSighting("Giles", "Corridor")
    without BadgeSighting("John", !"Corridor") as cB;
match cA before cB within 30 mins as cAA;
match componentOf(cAA,0) follow TerminalLogin("John", ?, "Norton") as cT;
interval componentOf(cT, 0), componentOf(cT, 1) as iReplayInterval;
```

Space constraints dictate that further examples cannot be given here, but the above examples are sufficient to indicate that by joining the expressive power of **OQL** with the temporal constructs of TEQL, the combined language has sufficient expressive power to enable a large collection of powerful queries to be carried out.

## 5.6 Summary

The event information *deposited* into an event repository needs to be accessed, reviewed, and retrieved in different ways that depend on the application and the activities represented by its event information. This chapter has reviewed the issues involved in providing the interfacing capability required of an event repository for it to service the diverse application domains where it can be deployed.

The temporal formalism adopted by the treatment was defined. Subsequently, the properties and temporal characteristics of a suitable event query language were determined. After reviewing related work on temporal languages, the remainder of the chapter presented a query language called TEQL.

This language is designed to satisfy the query requirements of an event repository, and allows the representation and treatment of qualitative temporal relations between events, composite events and temporal intervals. It also supports the creation of derived views of the original event history in order to aid interpretation, analysis and review.

# Chapter 6

# The Repository Architecture

This chapter presents a design for the architecture of a general-purpose event repository service that addresses the requirements of the application domains identified in Chapter 2. It also integrates seamlessly with the HERALD event notification infrastructure described in Chapter 4.

Section 6.1 starts by identifying the functional requirements of an event repository and Section 6.2 then introduces a specific design for an event repository that meets these requirements. In order to offer the flexibility required of a general-purpose event repository, the architecture proposed is functionally divided into what are termed the *repository server* component and the *repository client* component.

Section 6.3 looks at the role and structure of repository clients, and discusses their exclusive view of events, their interfacing with a repository server, and how they can be written.

Designing the generic aspect of the storage architecture, i.e. the repository server, is addressed in Section 6.4. Approaches to providing the required functionality are discussed and the modules that support client interfacing and query handling are introduced. In particular, Section 6.4.4 identifies the characteristics required of a storage sub-system for event data, and then proposes a custom log-based hybrid design.

In Section 6.5, the issue of propagation of event histories in between event repositories is considered.

In this chapter, the term *conventional database* is taken to refer to an industrial relational or object-oriented database system.

## 6.1 Functional requirements

Based upon the discussion in the preceding chapters, one can identify a number of functional requirements of an event repository. In particular, one needs to:

- support storage and retrieval of event information, where an event is defined as being a message denoting the occurrence of an activity of interest, of a defined type and structured with typed attributes.
- support high-speed writing, so that high volumes of events can be written in quickly without data loss.
- enforce the fact that event information has a historical value, so it may not be modified once *deposited* (entered) into storage.
- support the temporal retrieval and replay query interface TEQL as described in Chapter 5. Since this is a superset of **OQL**, legacy applications can also query event data as if the repository were a conventional database.
- support the Object Data Management Group (ODMG)'s data model for primitive types [CB97] for attribute typing within events. This supports the operation of mirroring the contents of an event repository into an ODMG compliant conventional database management system should this operation be required.
- provide a repository that can be integrated seamlessly within an event-driven infrastructure like HERALD, where event components can interact with it: either implicitly (transparently) through registration policies, or explicitly for carrying out specific queries on past activities.
- support application-driven contextualisation of events of different types through the notion of *sessions*.
- support derivation of new higher-level views of the event data.
- provide for interaction between event repositories for the purpose of replication or merging of event histories.
- support evolving event types through schema versioning. Allow operations to be carried out on events with schema that are no longer current, i.e. have evolved to a different specification.
- allow multiple applications to access it simultaneously, and preserve the view of each individual application by distinguishing between the event information each is aware of and can carry out operations on. Ensure that such a view is consistent to each application – i.e. it only sees coming out of the repository what it deposited in the first instance.

It is difficult to satisfy these conflicting requirements with a single monolithic architecture. Any general-purpose solution must be flexible and dynamic in its composition, so that it can be adapted to the specific requirements of any one application.

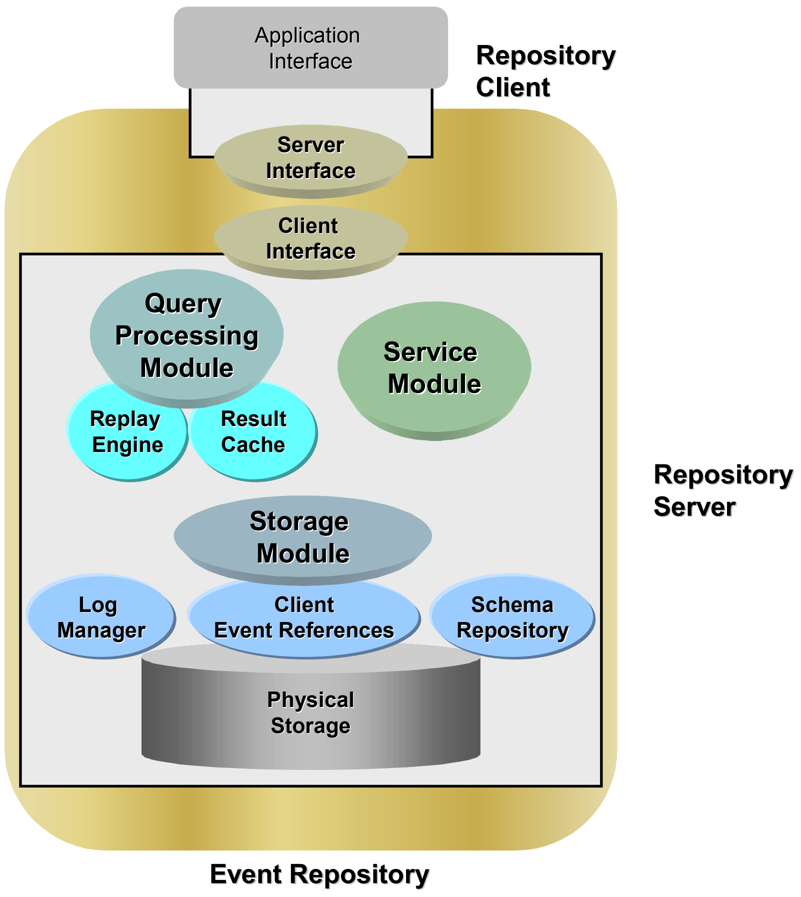

**Figure 6.1**
The structure of an event repository

## 6.2 Architecture overview

This section proposes a specific design for an event repository that addresses the functional requirements listed in Section 6.1. In order to offer the flexibility required of a general-purpose event repository, the architecture proposed is functionally divided into what are termed the *repository server* component and the *repository client* component.

In essence, every instance of an event repository consists of several distributed components working concurrently and co-operatively, primarily a number of *repository clients* and a *repository server*. While the server component of the architecture is kept as generic as possible to be of use in different scenarios with diverging requirements, a repository client can be tailored to be highly specific to an application. The actual storage and retrieval functionality of the event repository is supplied by the repository server.

The aim behind the design of this architecture is to provide a core set of generic services coupled with highly and easily tailored components, so that the whole can

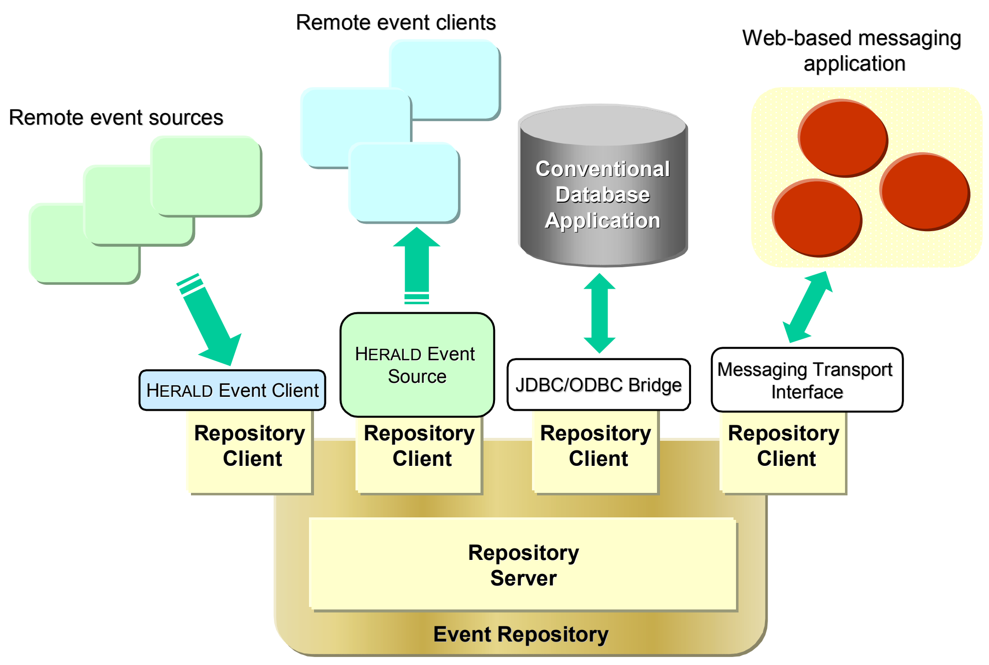

**Figure 6.2**
Interfacing to the event repository through customisable clients

provide the required facilities without compromising on performance. Figure 6.1 illustrates this functional arrangement. The functional modules identified within this figure are described in Sections 6.3 and 6.4.

The distinction between a repository client and a repository server is conceptual, and within some applications, it is possible to integrate the two components into one software entity. One such case is the embedding of an event repository within an HERALD *event source* or *event client*, as described in Chapter 4.

Figure 6.2 suggests some application domains that can be interfaced to the event repository through the defining of a custom repository client. These are distributed applications built with HERALD or some other similar general-purpose event notification infrastructure, conventional databases through an **ODBC** or **JDBC™** bridge, and wide-area message-driven systems.

Section 6.3 will now describe the functionality encompassed within the repository client, while Section 6.4 then discusses the internal structure of the repository's server component and the issues relating to event storage and retrieval within its modules.

## 6.3 Repository clients

A *repository client* component acts as the application writer's interface to the event repository. While the *repository server* component is generic for all applications and event types, the repository client can be highly specialised and tailored for the particular application being serviced. The client interfaces with the server through an

Application-Programming-Interface driven through remote-method-invocation. Through this interface, it can instruct the repository server to create a new event view for it, pass it event schema, deposit event streams, configure storage and archive parameters, and pass TEQL queries that return events to or initiate event replays into it.

### 6.3.1 The nature of repository clients

When an event repository is embedded into a HERALD event client, the event client implements the client-side of the repository. The client registers interest with one or more event sources in order to obtain the particular events to be stored. The events notified and stored can be of different types with differing schema, and will reflect the specific registration templates the client will have made with those event sources. Figure 6.3 shows some examples of this usage within a HERALD environment.

The repository server requires an event type's scheme before it can accept any instances of that event. A HERALD client can obtain event instances' schema from their respective sources and pass them on to the server. A client that is interfacing with another messaging infrastructure, however, might need to obtain the scheme in a different manner. In particular, if event messages within that infrastructure are self-describing (for example, using **XML**), the client has to extract and generate an event scheme from the messages, and define a range of types against which messages can be identified.

As schema versioning is supported, the scheme supplied must also be qualified with the version identifier of that event type. This structured version identifier is particular to the repository's server and client components, and is consistent with the schema versioning model used in the rest of HERALD. When the repository client is interacting with alternative infrastructures, it needs to configure the schema itself.

Since the repository endorses the ODMG data model, it may therefore be necessary for the client to carry out type mapping if the type space of the messaging environment being used is different. This would have to be a bi-directional mapping.

For applications that are not part of an event notification infrastructure, the repository client acts as an interface wrapper to the repository's functionality. Since such applications tend to be more tightly coupled than the more dynamic applications within event-driven infrastructures, it is likely that a client could be written that 'knows' the structure of the application being supported. This means it would be aware of the format of any internal messaging information it uses and of the whereabouts of the sources and sinks of such information. The repository client would therefore be able to define the schema of the data to be captured, and supply it to the repository's server component; as well as carry out any bi-directional mapping required to format the data. The client would then interact with the application's centralised or distributed components in whatever proprietary manner is required, be it through remote method invocation, messaging, or through an object request broker. Examples of such applications tend to be specialised environments

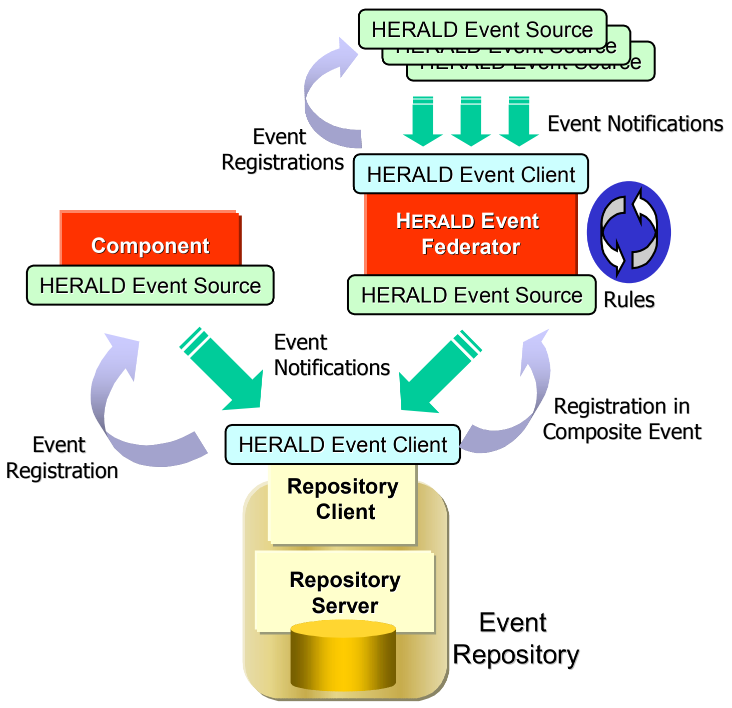

**Figure 6.3**
Examples of the event repository interfacing with a HERALD event source and a HERALD federator.

like fault-tolerant infrastructures, thin-client applications (see the case-study in Section 7.4), and federated and distributed databases.

In this latter case, given that the number of processes are known and there is more symmetric interaction, the repository client would also be able to implement mechanisms that enforce global causal ordering of events. Using vector clocks the application could implement a global logical time, which the repository could map to real time once it has ordered the events being received. The best way to carry this out is to provide a sliding temporal window of observation at the client, into which incoming events are placed until delayed messages can be received, then deposited into the event store in the right order. With logical ordering, however, events' temporal separation is lost.

In case of a HERALD application, event timestamps correspond to real time at the originating source, since the linear increase in size of a logical timestamp with the number of event components makes vector clocks unacceptable, and given the lack of global application structure, probably simply not possible. Therefore, the application has to assume the presence of an external clock synchronisation protocol like **NTP**. Nevertheless, if the application is sufficiently dispersed over a wide-area, network delays might need to be taken into account. In this case, the HERALD client might also have to insert a sliding temporal window of observation between its receiving events and depositing them, setting the length of the window to be such that network delays can be accommodated. Since HERALD timestamps reflect *real time*, once event exit the observation window correctly ordered, they can be deposited into the repository with the original relative separation as they occurred.

The issues highlighted in this discussion: event schemes; type mapping; countering unsynchronised clocks and network delays; are representative of a host of application specific issues that do not apply globally and are thus addressed within the repository clients provided by applications.

## 6.3.2 Event handling

Each repository client sees a segregated *view* of the event information stored within the repository server. When a repository client first communicates with a repository server, it is allocated a special client identifier and an authentication key, which it must then present whenever it re-establishes communication with the server. This ensures that only the client that has stored specific event information, or a specific view of that event information, can retrieve it. Once the client starts getting event notifications from the event sources it had previously interacted with, it then opens a stream connection to the server and *deposits* those event instances.

The client defines *sessions* with which to tag the event instances being deposited. As discussed in Section 5.2.5, an event instance may belong to one or more conceptual sessions, and a session can contain within it events of all types. The advantage of defining sessions is primarily evident when it comes to retrieval or replay of events, as the client may then restrict its retrieval expressions and queries to the context of a session. Therefore, a session is essentially an index whose semantics are determined by the application.

When querying the event data, the client can request the creation of new sessions that present some higher-level interpreted view of the event history. These sessions are known as *derived sessions*. They differ from regular sessions in that they can be modified and built up gradually. Derived sessions are persistent until the client asks for them to be deleted.

## 6.3.3 Interfacing with the repository server

A client can have multiple simultaneous connections to the repository server, enabling it to pass in multiple event types concurrently. It can also be depositing and retrieving event information from the server simultaneously, although the concept of present time, i.e. "now", would be continuously changing as more events are deposited by it into the repository. This is of relevance in open-ended *correct-time* replay operations, where the term correct-time denotes an event replay at the same temporal speed as the original event depositing. The client therefore acts as both the input and output interface of the event information stored in the repository and belonging to an application, a user, or their agent.

A repository's client and server components can be co-located within one execution space, or can be located on different machines, in which case they interact through a remote-method invocation interface.

As already mentioned, when a repository client first opens a connection to a repository server, the server creates a *repository view* for that client. All the operations invoked on the server by the client will apply only to that segregated view of the event information stored in the event repository. The client packages TEQL queries and replay requests within method invocations and passes these on to the repository server, which then responds according to the query. In case of a replay request the server will establish a stream connection with the client and start passing it the event instances forming part of that replay. The direction in time and relative speed of the event replay depends on the parameters passed within the TEQL query. It is possible for a client to terminate such a replay at any moment.

Outside of a messaging environment, an application might need to extract the event information stored in the repository in a more conventional way. A client can interact with the repository server through standard **OQL**. Through this interface, the repository appears as a read-only database where events are represented as records in the relational sense. This technique is particularly suitable for extracting events from the event repository and transferring them to a conventional database for further archiving or analysis.

## 6.3.3 Creating clients

Writing a new client to support the event storage and retrieval requirements of an application or event infrastructure has to be straightforward. In the prototype implementation of the event repository, the essential repository client's functionality is provided through a package of **C++** or **Java™** classes. These enable a client to be written very rapidly and the application writer can then focus on tailoring the client to the requirements of the application. The remainder of the functionality of the client within the event/messaging infrastructure depends on the application.

Within a HERALD infrastructure, the client can present itself to the application's distributed active components as an event source by integrating within it the event source libraries (as shown in Figure 6.4). If desired, this configuration can be used to abstract away the whole event repository functionality in the guise of an event source that initiates event instance notifications from its event store based on some

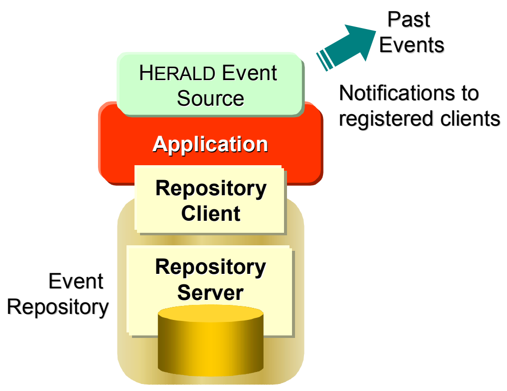

**Figure 6.4**
An application that carries out analysis on event histories can interface with a HERALD infrastructure by exporting an event source interface

application criteria. Likewise, the client can be a HERALD *federator* by implementing an event client interface, an action injection interface, and a rules module; an event *gateway* by supporting both the event source and client interfaces; or even a *broker* by integrating the brokerage processes. Each HERALD interface being like a pluggable building block implies that several permutations are available to the application writer. One can thus integrate event repository functionality within an application in a variety of ways, both implicitly and explicitly.

As described in Chapter 4, each HERALD event source can optionally integrate an event repository within it, as the source libraries define a repository client within them. The purpose of this client, if enabled, is to capture each event occurring at that source, so that in addition to real-time notification of that event to registered clients, a persistent copy can be retained. The enabling of this feature allows that event source to support a number of registration policies requiring persistence. This is illustrated in Figure 6.5.

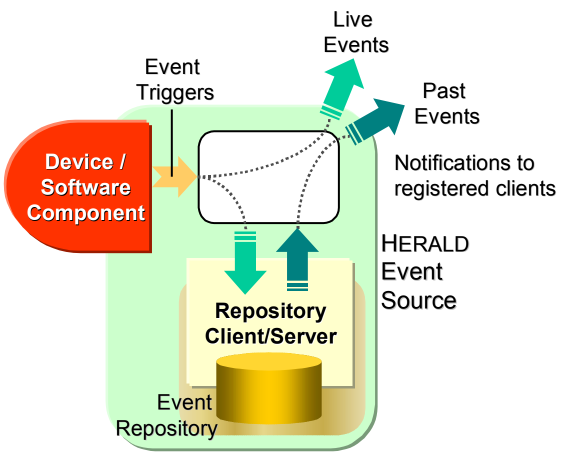

**Figure 6.5**
Embedding an event repository within an event source.

In similar fashion, HERALD event clients can also embed an event store in order to retain a history of the events they consume.

## 6.4 Repository server

The repository server is the generic part of the repository architecture. While the client can be customised according to the interface requirements of the application domain it is addressing, the server remains largely unmodified. It can, however still be customised by enabling or disabling a number of functional modules within it.

There are three primary conceptual modules within the repository server:

- the *service* or *interfacing layer*, which handles communications with the repository's application clients and allocates resources like spawning handling threads,

- the temporal query processing engine, which parses, breaks-up, optimises and executes temporal queries in TEQL,
- and the *storage layer* or database sub-system.

Although there are numerous possible architectures for integrating these modules within this server side of an event repository that satisfy the functional requirements listed in Section 6.1, two broad approaches can be distinguished: *layered*, and *built-in*.

The next two sub-sections discuss the general architecture of the repository server, while the following sections discuss several specific issues involved in implementing the modules making up the repository server.

## 6.4.1 Designing a storage architecture

There are two approaches to providing the service required of an event repository: the *layered* approach, and the *built-in* approach. These will now be discussed in turn, and a synopsis is then given. Figure 6.6 illustrates these two approaches.

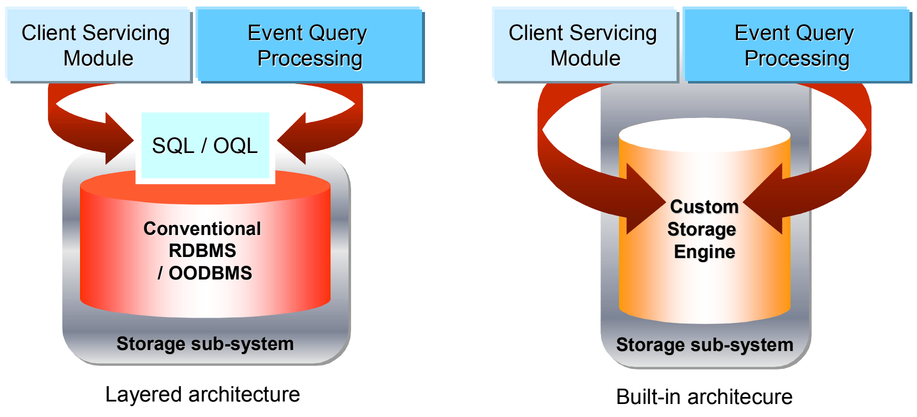

**Figure 6.6**
The layered approach (left) and the built-in approach (right).

### The layered architecture model

In a layered architecture, all event aware components reside in a module built on top of a conventional database system. This architecture can also be described as a *loosely coupled* system, since temporal query processing is completely separated from the database system. In this approach, temporal operations, such as those proposed in Chapter 5, need to be defined in terms of the query language supported by the underlying database, this probably being **OQL** or **SQL92/SQL3**. Likewise, all storage and retrieval operations are carried out through the database management system's query interface. Events need to be mapped to relational records or objects, depending on the nature of the database management system used. Higher level event-oriented functionality like event type inheritance, schema versioning, and event sessions, need to be explicitly defined using tables and views. Replay of event streams requires running an appropriate **OQL/SQL** query to identify the data that will participate in the replay, buffering that retrieved event data in memory, and then notifying the event instances with the same temporal separation as defined by their timestamps from that buffer.

The advantages of a layered architecture are:

- A conventional database system can be converted into an event repository without modifying the database system at all.
- The conventional database system abstracts away all the underlying storage complexity such as transaction management, lock management, indexing, storage management, and fault-tolerance (although this may be unnecessary overhead).
- Organisations often have significant material and human resources invested in maintaining and running one or more database management systems. Considerable investment, experience in its use, and trust in the database's dependability, can offset performance and functional considerations.

The disadvantages of a layered architecture are:

- There is potential for poor performance, since substantial communication overhead may be required between the module that provides temporal querying and the database system.
- Since there is no direct access to the storage subsystems of the conventional database system, all event temporal functionality needs to be mapped to the database's query/command interface. Some operations will therefore require substantial temporary memory buffers to execute.
- Since the event temporal functionality module cannot interact with the subsystems of the conventional database system (such as the transaction manager, lock manager, etc.), certain features that require access to these subsystems may not be supported (such as concurrency control for simultaneous insertion of event streams).
- The conventional database might not be able to perform fast enough to cope with high-speed writing of event instances into its storage system. The standard overhead of heavyweight database functionality, like transaction management, affects performance even when such capability is not required.

## The built-in architecture model

In a built-in architecture, all event temporal functionality components become part of the database system itself. This architecture can be achieved by modifying an existing conventional database, by using a database system toolkit for conventional features while adding temporal event functionality, or by building an entire event repository from scratch. A built-in architecture can be termed the *tightly coupled* approach, since temporal event retrieval operations and processing are directly integrated into the database system. In this approach, the underlying storage can be directly tailored to the structure of event information. The storage can be optimised for the specific nature of event data. Due to its historical value, such data need not be modified once stored (although it may be archived). Indexing of the event data is carried out based on the timestamps of the event data and according to the real-world timeline represented by the stored events. The temporal rule processor evaluates rule conditions and locates event sequences by performing operations directly on the database. It is possible for the rule processor to exploit the underlying storage arrangement of the data and the database system's query evaluation facilities.

The advantages and disadvantages of a built-in architecture are exactly the converse of those for a layered architecture. The advantages are:

- The database storage subsystem can be extensively tailored to support the specific nature of event information and the temporal queries that may be carried out on it.
- Event sequence identification, pattern matching, filtering and replay can be performed efficiently since they occur directly within the database system.
- Access to database subsystems allows implementation of sophisticated features for rule evaluation, concurrency control, archiving and error-recovery.
- Conventional database functionality that is not required and that would incur an unnecessary overhead can be disabled or removed completely.

The disadvantages are:

- The implementation effort can be substantial and may require modifying existing code at best, or writing a complete underlying database system.
- If different conventional database systems are converted into an event repository, it is likely that differences between the conventional database systems will carry over into the event-oriented components.

## Synopsis

In the layered approach to building an event repository, it is possible to use a conventional database management system for the actual low-level storage of event instances. The most significant advantage of this approach, as illustrated in the previous discussion, is the ease of implementation. This approach, however, ignores the specific nature of the data being stored.

The ideal design is one that is tailored for capture of high volumes of multiple streams of relatively small data objects that need not be modified as they are to constitute a historical record. This renders most of the functionality of conventional database management systems irrelevant, and experiments [Nel98] indicate that even high performance commercial databases struggle to cope with the high-volume of event deposit required of an event repository service. Within distributed debugging systems, where there is considerable experience in event tracing, simple logs are still the preferred means of storing events persistently.

Having discussed the possible architectural approaches to providing the functionality required by an event repository, it is evident that a tightly coupled approach meets the primary performance requirement considerably better than a loosely coupled approach based around a conventional database management engine. It is possible that with very fast hardware this performance issue may become academic, but computing trends indicate that although the hardware may become increasingly more powerful, it is likely that data throughput requirements will similarly increase.

Sections 6.4.2– 6.4.4 illustrate the functionality of the main conceptual modules of an event repository within the context of a tightly coupled built-in architectural model.

## 6.4.2 Service module

The first modular layer of the repository server is the **Service Module**. This module receives command queries and event streams from the repository clients. In this communication model, instructions and queries are packaged as parameters and passed as messages from repository clients to the **Service Module**. This allows the **Service Module** to prioritise operations and allocate threads. This module carries out authentication of the repository clients, allocates and establishes client identifiers for denoting each client's view of the repository, and transparently tags each command query with this identifier.

In order to be able to handle multiple repository clients concurrently, the prototype implementation of the **Service Module** allocates a thread to handle each client, and that thread can then spawn further threads to handle simultaneous connections with different command queries from that client. This enables multiple input and output streams to be handled concurrently. Output operations are given lower priority than event input operations.

## 6.4.3 Query processing module

The **Query Processing** module analyses incoming queries and according to their nature, input or output, allocates resources appropriately. Command queries can be categorised as follows:

- *Command operations*; e.g. establish a client/server relationship and obtain an access key, define an event type and specify its scheme, add an updated version of a type's scheme, set archival and storage thresholds, configure the size of the result cache, make derived sessions persistent, flush all indices and events pertaining to the client.
- *Deposit operations,* deposit an event instance into the repository or open a typed channel for continuous deposit of multiple event instances.
- TEQL *retrieval queries*; e.g. locate composite event patterns, locate event intervals, identify temporal spans, locate individual events according to some temporal criteria. It is the nature of TEQL queries that most retrieval operations do not return a definite result, but rather a set of results matching the criteria specified in the query. Therefore, TEQL 'retrieve' queries return a structure that contains a series of results matching the query. The result sets can contain primitive events, composite events, or intervals. A reference identifier is attached to each result set

since these are retained in memory for some time, in case the same set of data is to be reused in a subsequent replay or retrieve.

- TEQL *replay queries*; e.g. locate some temporal interval whose end-points are denoted by some temporal value or event timestamp, and replay the events that occurred within its timeframe (or all types, of some type, or with some more specific filtering applied) at the original replay speed or some fraction/multiple of it. These open an output stream that has to be handled by an application-defined handler method in the repository client.
- *Referenced data replay queries*; these are not TEQL invoked queries but rather a method-invoked replay where a result reference is passed as parameter. This reference identifies a set of events, or an interval, that were the result of an earlier TEQL retrieve query.
- **OQL**-*only standard queries,* for these operations the event repository appears as a read-only conventional database.

The query layer identifies each query or command, and checks it for correctness. If it is a TEQL query it is parsed and validated. The query is then executed and the results sent to the repository client through the **Service Module**. A result could be a batch of events, or a stream of events from the **Replay Engine**.

The **Replay Engine** buffers a resulting set of primitive or composite events, and replays them out at the same temporal separation as denoted by their timestamps.

The **Result Cache** retains the most recent query results in case further queries are to be executed on them, as well as the most recent queries' syntax. This is most useful in applications where a retrieve is first carried out, and then the user is allowed to select some sequence of events for replay from the set of sequences returned by the query. Obtaining the sequence from the **Result Cache** is more efficient than re-evaluating the original query. Result sets returned to the repository client are tagged with a reference that can be quoted in TEQL queries in order to force retrieval from the cache. If the data has expired, the query syntax is retrieved and re-executed.

Both the **Replay Engine** and the **Result Cache** can be disabled upon creation of an event repository, in which case some functionality will be disabled. This will trigger error notifications to the client should their services be requested.

Whether the query layer implements TEQL queries directly by interfacing with the storage sub-system (discussed next), or by mapping TEQL queries to **SQL/OQL**, depends on the choice of the storage sub-system and on the query interface it provides. For the prototype implementation of the event repository a custom storage sub-system was developed. Interacting with this involves interfacing to a log management module.

## 6.4.4 Storage module

This section proceeds from the observations made in the synopsis of Section 6.4.1. It further qualifies those observations by identifying in detail the storage requirements of the storage component of an event repository, and then presenting a custom log-based design to address these requirements.

## Storage requirements

Based on the discussion in the preceding chapters, the functional requirements of the storage component of an event repository can be summarised as:

- *High performance writing* – The storage system needs to be able to support and handle the writing of several hundred events per second, where the events can be of different types (of differing and evolving schema).
- *Performance and redundancy vs storage* – These have a higher priority than volume of storage. Trends over the last ten to fifteen years indicate that while storage capacity on magnetic and optical media keeps doubling periodically and decreasing in cost simultaneously, access speed and data throughput have improved only slightly (in relative terms).
- *Event depositing and archiving* – Event instances are always deposited into storage and not modified thereafter. As the store cannot grow indefinitely, archiving of earlier events needs to be provided.
- *Event retrieval* – Access is required to each stored event instance and to stored sequences of events in order of storage timestamp. The sequence can consist of event instances of the same type or different types.
- *Appropriate fault-tolerance* – Should the machine crash, or a disk block failure occur, minimal event loss will occur and existing logs are not damaged.

These requirements differ from those traditionally bounding a relational or object-oriented database. This fact has been recognised in several projects looking at retaining events relating to process behaviour for distributed systems debugging [Han88, Tsa83, IYS86, LS90, NX93], where logs of event traces are the preferred approach due to the high-speed of writing they permit.

## Storage system configuration

In order to address the requirements listed above, a log-oriented storage system is proposed for storing event instances.

The system consists of a **Log Manager** that handles the underlying storage log files. Event instances are categorised by their type, and distinct types are stored within different logs. This enables keeping all the entries in a log to the same size. Logs are duplicated for reasons of fault-tolerance and consist of several 'page' files, each containing an atomic number of event instance entries. The log duplicates are written to alternate disks if available. Multiple threads handle concurrent writing of multiple logs.

Writing only involves opening one page file from an event type log, and the write is duplexed to the mirror of that page file. An event type log page is buffered in memory until it is full, at which point both the primary and secondary log page files are closed. All but the last event instance can be read without locking, and the last event instance cannot be read while it is being updated. A lock protects this write operation.

Session indices are stored in a similar fashion, with index entries identifying the event instance type, and its relative position within the appropriate log. Session indices are generated in real-time during logging of event instances, and the same locking constraints apply to them as to event instance logs.

A **Schema Repository** retains all the event schema of the event types currently deposited into the event repository, as well as the schemas of any derived composite events referenced by derived sessions. In addition, a **Client Event Reference** is retained for each repository client. This details the repository view that applies to that client, including the event streams and sessions it has access to, and references schema of all its event types. When a repository client establishes a connection with the repository server, this data is read into memory.

## Log Manager

The **Log Manager** (see Figure 6.7) provides a clean and protected interface to the underlying log files. It abstracts the details of multiple log pages, duplexing, locking, and archiving to tertiary storage. It carries out fault tolerance through the use of 'careful writes'.

It keeps track of the segregated event information space according to the repository clients, allocates logs for each event type version within that information space, and generates a number of index files for accessing the event instances within that log. The log is partitioned into log pages, and each log page is uniquely identified.

The **Log Manager** maps the event logs into a growing collection of sequential files provided by the operating system, the file system, and the archive system. As a log fills one file, another is allocated. The size of files represents a compromise between performance and fault-tolerance. It is faster to work with larger files, but they are more likely to get corrupted. The archive system is needed because event logs grow without bound, therefore, in general, and based on the repository client's preferences, only recent events are kept online.

The **Log Manager** maintains a storage structure table that details all the log tables, their composition into log pages, and mapping to physical operating system files. It also reflects their mirrored duplicates. When log pages are archived to slower disk devices, it amends its physical location entities to reflect that access of these tables has to be carried out through the **Archive Manager**. Since events are ordered according to their timestamps within the event logs (and log pages), the structure table also contains the temporal interval covered within each log page. This information is important in assistance of fast access to an event by its timestamp, or by its temporal relationship to other events or event sequences.

During a query, the **Log Manager** handles opening of physical files and the granularity of the locking to be used. By default, the **Log Manager** reads in event instance entries and opens page files optimistically. This is of most relevance when

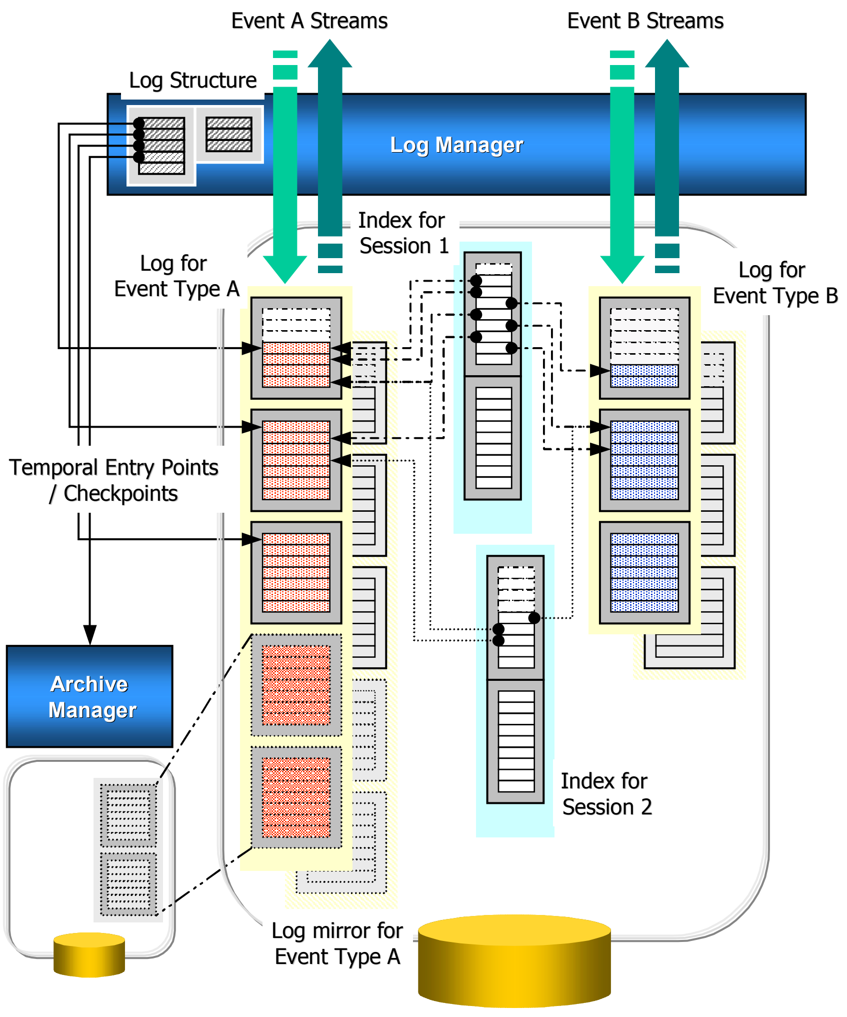

**Figure 6.7**
Structure of the storage module, event files, session and temporal indices

some of the events to be retrieved occur in old and currently archived log pages, as real-time de-archiving is a slow operation.

## Naming

An important module within the storage sub-system is the **Naming Manager**. The purpose of this module is to define the physical mapping of event log tables and their log pages to actual physical files, and define the names of those files.

## Indexing

Since the storage architecture is optimised for rapid writing, read performance is primarily achieved through indexing.

All events are stored sequentially in the same order as they are received by the event repository. This is termed the *deposit timestamp*, and is different from the *creation* *timestamp*, which is a property of all event types set by the event source from where the event originated. The deposit timestamp is added to the event instance and does not replace the entry for the creation timestamp. In addition, a unique *event entry number* (EEN) is attached to the entry.

Session indices are created in real time while the events are being deposited. Each entry in a session index consists of a reference to the event the entry points to, with the reference taking the form of a reference to the event log, the page within the log, and both the relative position of the event instance relative to the beginning of the log page as well as the timestamp of the event.

Access by absolute values of real time, as well as by intervals of time, is required by a number of queries that can be run on the event repository. In order to facilitate this, the **Log Manager**'s representative structure contains, apart from physical information about each log page (like its filename), its temporal entry point; i.e. the timestamp of the first event instance stored into it. This neatly partitions the event log for an event type into ordered chunks of time, thus enabling fast read access and search by timestamp or absolute time value.

### Archiving

A sub-module within the storage system is the **Archive Manager**. This picks old pages from within event logs and *archives* them. Two ways of archiving are suggested in this discussion:

- *relocation of old pages to some form of slower storage.*
  This is likely to involve compression of the pages and relocating to a slower disk, like a network-mounted file-system.
- *moving of older events to a remote event repository.*

The operation of archiving is carried out within the **Archive Manager** by a low priority thread based on the storage thresholds set by the repository client for which event information was deposited.

A common type of failure within an event repository could be running out of disk space. The **Archive Manager** counters this by periodically monitoring the amount of free disk space left on the current media, and increasing the priority of its active archive thread accordingly.

Since sometimes retrieval query operations need to be carried out on event data in archived files, real-time de-archiving is required. For this reason, the **Archive Manager** must reserve an amount of storage space on the local high-speed medium so that it can fetch and decompress log pages from the archive location and place them in this 'buffer' space.

## 6.5 Propagation of event histories

There are a number of reasons for why an event history kept at one event repository might need to be moved elsewhere:

1. The event repository where the data was captured might be running on a platform with limited resources, and therefore the repository has been instantiated with important modules disabled. A remote, fully functional repository can enable a fully featured analysis of the event data in a way that is not available at the original event repository. In this case, the whole event view of the local repository client needs to be propagated to a remote repository server.
2. There might be insufficient storage space on the local repository so event data has to be archived at a remote event repository. In this case, only events that are older than an application-defined threshold are propagated.
3. Merging of event traces collected at local event repositories can address the problem of lack of global time and of excessive network delays. In this scenario, separate local event sequences are stored on the same machine where they are generated, using stable unsynchronised clocks. By measuring the rate and offset of the individual clocks, the timestamps in the event histories are adjusted at a third remote event repository to reflect a consistent, or global time base. The adjusted event histories are then merged into an accurate global event history.

In all cases, the repository client's access key to the view on the repository server must be transferred to the remote server. The local client then has to propagate this key to the relevant remote client that intends to access the remote copy of the event data.

In order to support this, event repository servers provide a **History Propagation Interface**. Through this, event repositories can communicate in both directions.

In cases 1 and 2 above, the propagation is driven by the client at the original repository. This makes the event repository server communicate with a remote counterpart, and request of it that it establish a remote event view. It can then transfer batches of events directly into that view. This bypasses the deposit timestamping mechanism at the remote server so that the 'now remote' history retains the original deposit timestamp. The remote event view will be appended on subsequent transfer operations.

In case 3 the propagation is controlled by the client at the third (the remote) repository. This first needs to get the access key of both the repository clients that control the local event histories that are to be sent to it. It also needs to determine the $\alpha$ and $\beta$ constants for the clock at each local repository. In effect, $\beta$ adjusts the rate of the local clock and $\alpha$ corrects for its offset with respect to the clock of the finished global history. Dietz [Die96] surveys and recommends a number of algorithms for obtaining these two constants. It then needs to make its event repository obtain the two event histories and trigger a merge function passing the $\alpha$ and $\beta$ constants as parameters. After this operation it can commit the new global history to a new view of the event data.

Figure 6.8 illustrates these two scenarios; cases 1 and 2 are shown in part A., while case 3 is shown (with less detail) in part B.

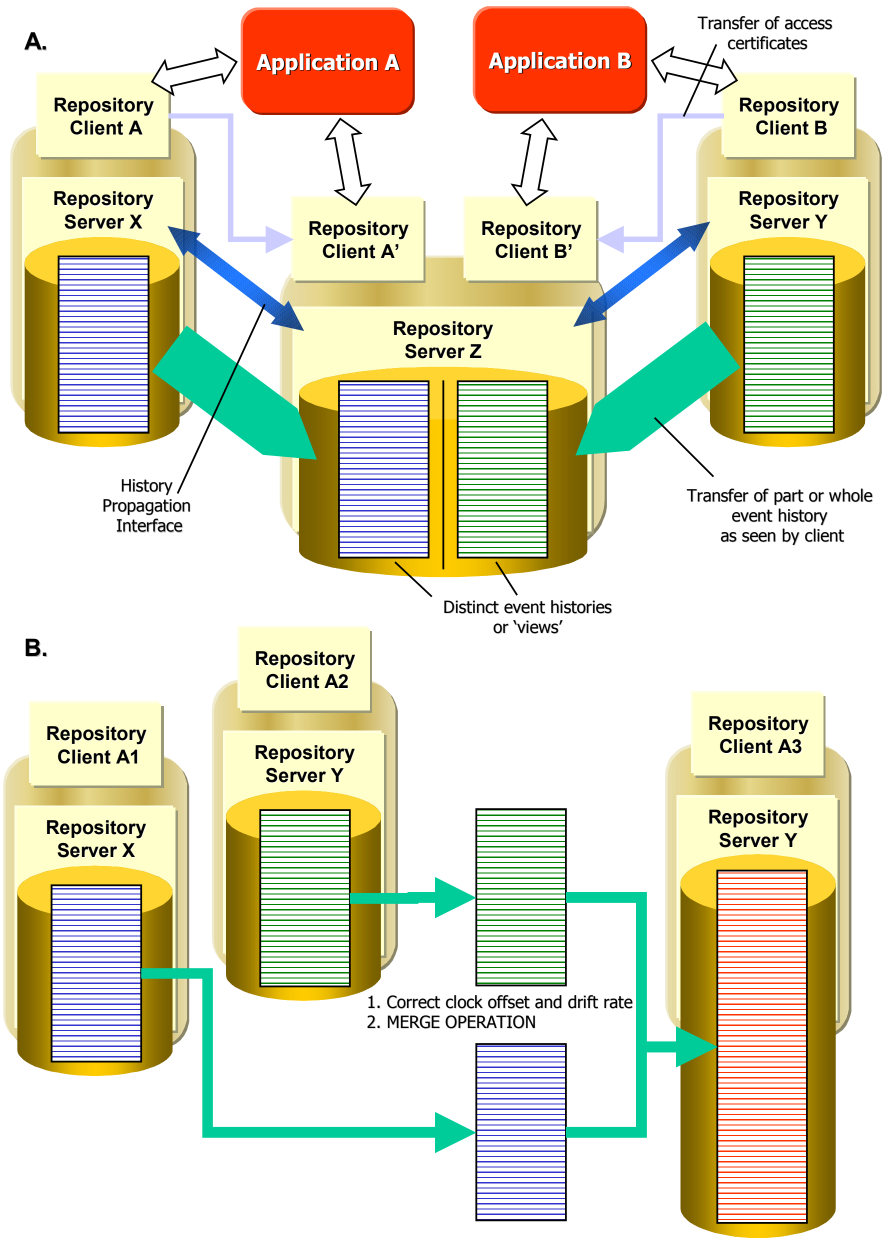

**Figure 6.8**

A. Propagation to remote event repository controlled by local client

B. Merging of two event histories into one globally ordered one, controlled by client A3.

## 6.6 Summary

This chapter has presented a design for an event repository architecture. This addresses the requirements of the application domains identified in Chapter 2, and can be seamlessly integrated with the HERALD event notification infrastructure.

The functionality required of an event repository was identified, and a specific design for an event repository that addresses this functionality was illustrated. The architecture proposed is functionally divided into what are termed the *repository server* component and the *repository client* component. The role and function of these two components were then described. In particular, approaches to providing the required storage structure within the server component were highlighted, and a storage design put forward.

The section concluded by considering the issue of propagation of event histories in between event repositories.

A prototype of the architecture presented was implemented. This enabled the architecture to be deployed in a number of scenarios. The prototype, although not as optimised as could be, easily met the incoming volume of events in these scenarios. Details of these case studies are given in Chapter 7.

# Chapter 7

# Experiments

This chapter introduces a prototype implementation of the storage architecture presented. It discusses some implementation-related issues and then proceeds with considering some of the ways in which an event storage service can be deployed within different application domains. It then illustrates four applications that were used as experimental case-study deployments of the event storage architecture.

These applications were chosen because of the breadth of scope they cover, and their distinct event storage and retrieval requirements; this being indicative of the wide-ranging applicability of the event notification and storage solution proposed in this dissertation. Some of these case-study applications were carried out in collaboration with other researchers and institutions, and this is indicated in the text where appropriate.

The experiments illustrated are as follows:

- The automatically generated diary (Section 7.3) illustrates the ease with which information can be gleaned from several sources and integrated using the event notification approach. The approach taken in this experiment enables locating information based on its temporal context, and enable limited analysis of working activity.
- The thin-client session capture tool (Section 7.4), built on **VNC** remote access technology, describes one way of monitoring user activity on a workstation, in particular their interaction with an application. This allows applications to be evaluated for the usability of their interfaces and feature-set, as well as assist in teaching within classroom environments.
- The multi-service network management infrastructure introduced in Section 7.5 addresses the complex issue of managing the various servers that a modern information provider needs to service scalable customer requirements. The solution proposed uses multiple event repositories that collect information on servers and then forward those events to more advanced event stores for analysis. Therefore, while live events enable service management and administration, analysis of event histories is employed to determine the effectiveness of management policies and to guide their evolution.

- Finally, Section 7.6 describes **CURVE**, a virtual mobility visualisation and collaboration tool that allows remote users to visit a virtual replica of a real building, and view and interact with embodiments of the real users working in and populating it. As well as enabling review of the interaction of these real/virtual users, the event capture and storage service deployed also enables replay of movement of the real users in situations like emergency evacuations.

While not being comprehensive solutions in their own right, these investigations explored the potential of event storage in various application domains in order to lay the technological test-bed for more advanced designs that can address the complex research issues identified in Chapter 2.

## 7.1 Prototype implementation

Prototype versions of the event repository and of the HERALD messaging infrastructure were implemented in the course of the investigation. These prototypes were built as proof-of-concept implementations and support the core functionality and features proposed in this document. The next two sections provide a brief synopsis of some implementation related features.

### 7.1.1 The HERALD event transport

HERALD has been implemented as a set of **Java™** class libraries. **Java™** was chosen for its ability to have the same binary code deployed and executed onto heterogeneous platforms. Two high-level classes, **EventClient** and **EventSource**, embody the functionality of an *event source* and an *event client*. By instantiating one of these classes within their application, developers obtain HERALD *event client* or *event source* capabilities. Various service handlers and multiple threads are launched by the instantiated objects, which the application code does not have to interact with unless it wants to. Communication with user code is carried out asynchronously by the user defining object handlers (that inherit from HERALD interfaces) to be invoked upon specific occurrences, examples of these being registrations, errors, acknowledgments, etc. Various modules, such as event storage within the HERALD transport, can be activated or disabled according to the initialisation parameters passed to the constructor of the **EventSource** object. Future revisions of the transport can increase the options available by adding to the set of policies supported. Since these are activated through parameter passing, the API would not be modified and current applications would not require rewriting. This approach allows for evolution of the libraries while retaining backwards compatibility. The prototype implementation of HERALD does not use **Java™** types for event parameters so as not to tie the transport and its applications to the **Java™** programming language or type system. Instead, the user calls methods that allow him to insert or retrieve ODMG types that are only then mapped internally into **Java™** types. All other message constituent data is transferred in between event-components in UTF-standard format. This ensures that

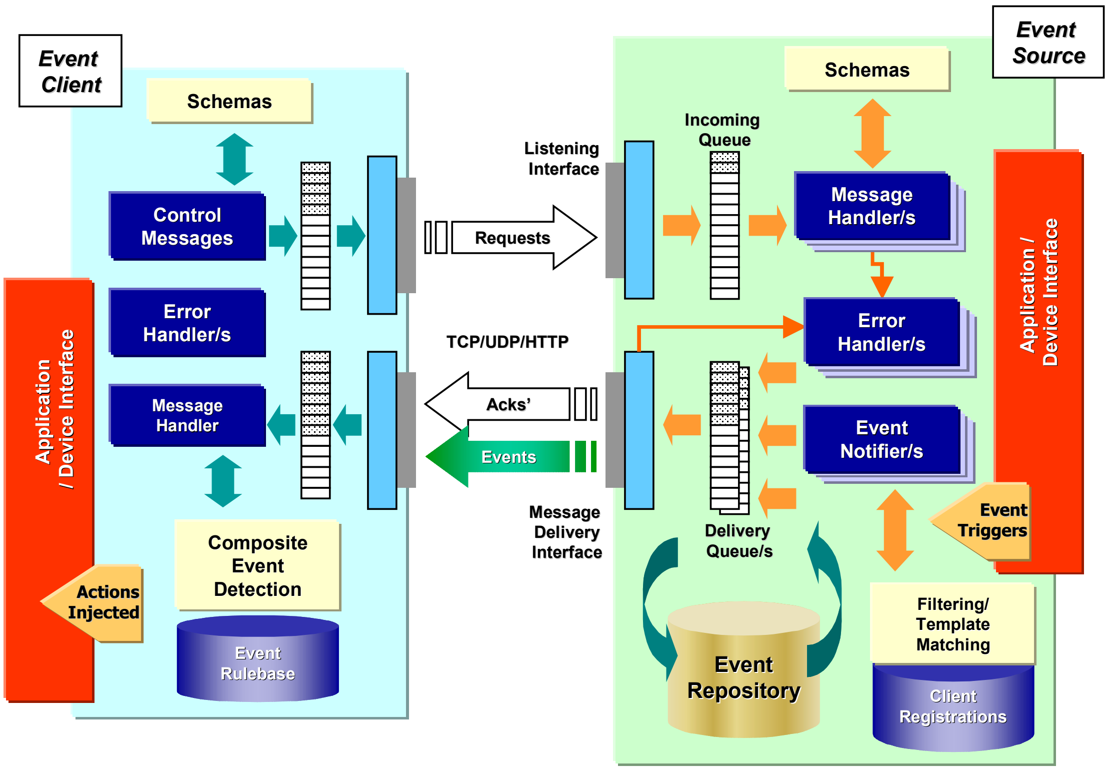

**Figure 7.1**
Structure of a HERALD event source and client

the transport libraries can be ported to other programming languages, and that these heterogeneous versions would be able to inter-communicate.

The event transport underlying HERALD makes no implicit assumptions as to the nature of the communications protocol being used, and abstracts the functionality of the latter through a defined interface. Events and control messages can therefore be propagated over both connection-less (UDP/IP) and connection-based network protocols (TCP/IP). Components can choose the transport they wish to employ dynamically, and can change it during the course of their execution if they so require.

All HERALD structured messages support and communicate the version number of the libraries that created them, enabling future revisions of the software to be backwards compatible while elaborating on the communications mechanism.

### 7.1.2 The event repository

Due to the importance of read-write performance for event storage and retrieval, a prototype event repository was implemented in the C++ programming language. This implementation provides for the core functionality required of the architecture presented in Chapter 6, and the retrieval interface described in Chapter 5. The server component of the repository is provided as a stand-alone implementation, while the client-side of the repository is provided in the form of a package of class libraries that an application developer can use to rapidly develop customised repository clients. These libraries are available in **C++** and **Java™**.

Communication with the server is enforced through client-side libraries, rather than allowing a programmer to communicate directly with the server, for security reasons. When a client establishes communication with the server and creates an event storage session, it receives a certificate that it must present whenever it re-establishes communication with the server for retrieval.

The server is fully multi-threaded, can handle multiple clients concurrently, and can be customised through a command line interface to provide (or disable) non-core modules that affect its performance and memory footprints. The client classes then determine which of these modules are available in any server instantiation to dynamically reflect the feature-set available to the application programmer. Typed and parameterised runtime exceptions are thrown if the developer's code is made to attempt invocation of disabled modules.

Each client session is allocated a time stamping thread that timestamps all events as they are received and buffers them for storage. Buffer sizes are monitored by another thread and grown dynamically in case the storage thread cannot keep up with incoming events. The storage threads create a directory to correspond to each client session, and keep separate log files for each event type and for session indices. Session indices are maintained in real-time during insertion of the event data.

The repository server maintains a **Schema Repository** containing the definitions of all the versions of the event types for which events are stored in it.

TEQL queries can be passed (as string parameters) through to the server interactively or may be pre-configured within the application code. This is convenient for when the client only carries out specific queries or presents a pre-configured graphical user interface with fixed retrieval and replay choices to the user.

Once a TEQL query is parsed and found to be correct, the composite event system starts accessing the event logs to locate composite sequences. This proceeds as follows:

- Locate bounding time interval over which the query applies. This may be the whole event history retained in the event repository, but is rarely so. In practice, queries are usually bounded within some values of real-time, or from some point in time until the present.
- Identify the starting event construct of the query, that is, the starting event for any composite match.
- Locate first starting event within bounding interval and scan from that point onwards for composite match.
- Depending on the consumption model specified in the query, new searches branch from the current point in the logs as events matching the composite sequence are located. These concurrent active searches are known as *evaluations*, and for each one a state evaluator is instantiated. The structure of these state machines is constructed and initialised upon parsing of the TEQL query.

The evaluation sequence implemented within the prototype is not optimal as it scans large numbers of event instances where it might be possible to optimise the access pattern to the data. This aspect is currently being investigated.

The query evaluator always retrieves all the primitive events requested by a query before returning them to the client. Similarly, event replays are evaluated completely before replay is initiated to ensure that the replay reflects the correct relative temporal separation and is not affected by waiting for any evaluation processing. This means that there can be a substantial pause from the client requesting a replay to the replay starting. This has been addressed by having the server first return a "query evaluated" result, upon which the client can then ask the replay to start proper.

For performance reasons, tight coupling between the server and client components is employed; therefore, by default the client code blocks in anticipation of server response. Asynchronous operation can be implemented by the developer by introducing multi-threading in the client-side application.

## 7.2 Deployment Configurations

Within the case study scenarios, several ways of deploying an event repository within a distributed environment were explored. The event storage service was provided:

- as a *system wide resource*, monitoring all event instances from one or more sources. An example is a centralised repository instance that captures all movement sightings from all **Active Badge**s in a domain. Application components can then interface with it to request a replay or analysis of these movements. The event repository then injects these active components with the movement events as if they were happening in real time. In this configuration, the repository can allow users to request retrieval of past sequences or replays from it according to some access policy. One such method would be for its interface to impose access lists on event types. One example case study (see Section 7.6) that uses this configuration is visualisation of mobility in virtual-reality for review of emergency evacuations. In this scenario, a centralised event repository injects events into the mobility visualising component.
- as a *user-centred service*. A user can instantiate a HERALD storage mediator and configure it to monitor all event types of interest. It is up to the user (or their agents) to configure the repository to organise events into sessions of relevance, which are useful for subsequent retrieval. Such a configuration can be used to monitor all activities the user or their agents engage in; an example being for automated diary generation (see Section 7.3). The repository could monitor all workstation activities in the background, receive notifications from system-wide services like the user's movement sightings and use of library facilities, and monitor their engagement in online collaborative activity.
- as an *application-specific service*. An active application can instantiate an event repository and use it for its own requirements. A debugging application can thus

use the repository to monitor system execution, allow analysis and review of the resulting trace and then destroy it once it terminates execution. In cooperative work scenarios, it might be desired not to keep a permanent trace of the interaction, but nevertheless use the event repository to support disconnected operation. One way of achieving this is to have the conference management objects instantiate their own event repositories, use them to provide replay for participants after reconnection following network partitioning, but then destroy them and all their event records once the collaboration terminates.

- as a *component-specific service*. An event repository can be embedded within a distributed component and be used solely by that component to provide persistent functionality. As discussed in Chapter 4, embedded event repositories can enhance the capabilities of event sources and event clients, as well as federators, brokers and gateways. In this configuration, the conceptual modules making up the repository can be tightly integrated with the modules of the event component in order to reduce unnecessary communication overhead.

These deployment scenarios are not mutually exclusive.

An important concern that applies in any scenario where replay is envisaged is that of *feedback*. Although one can capture the event messages that integrate an application, it is not always possible to then simply replay these back into the application to set its global state the same it has been at some point in the past. The reason for this is, of course, that in many components, the notification of an event will cause input/output to be carried out, as well as bring about generation of other new events that trigger further activity elsewhere. Therefore, replay can only be successfully carried out when it is possible to distinguish between externally sourced events, and internally generated events that are causally related to them. Only the former should be fed back into a replay, and even then, the non-determinism inherent within loosely coupled execution implies that the application might behave differently in this second replay. This issue has received prominent attention within investigations into fault-tolerant distributed systems environments and simulation [JZ88, SY85, WF92].

## 7.3 A ‘memory prosthesis’ -like diary application

A *human memory prosthesis* diary application was designed and deployed. The diary application illustrates the ease with which information can be gleaned from several sources and integrated using the event notification approach.

A ‘human memory prosthesis’, as defined by [LBC+94], denotes an application that aids memory recollection by supporting searching by temporal context. People often forget the details of a particular event or activity, but will recall the context of when that activity took place. Examples of this context are activities that took place before, during, or after the event in question. An automatically generated diary can serve this role in aiding recollection if it can:

- capture enough information about a user's daily activities, and
- present a meaningful means of querying the information.

In order to acquire reasonable granularity of information on a user's activities, information from both the physical and virtual (digital) dimensions a user works in was collected.

Users' physical location and movements were tracked through an **Active Badge** system [HH94], where each user wears a personalised infrared badge that transmits a signal every few seconds. This is picked up by widely deployed sensors throughout the building. The sensor network was interfaced to by an event source module. This module retrieved all badge sightings for all members of the department and kept track of when they changed location. It offered event consumers events pertaining to a user's badge sightings (typically every few seconds), clicking of the buttons on the badge, and movement events, generated only when a user is detected to have changed location. The **Active-Badge Event Source** module also encapsulated event storage functionality. Therefore, it could be interrogated as to the history of movement of any user.

Software sensors monitored a user's computer-based activities. In each of the cases below, a stand-alone component or a wrapper around some application, was deployed to monitor some software activity (see Figure 7.2). This generated HERALD events of one or more specific types when important operations are carried out in software. This experiment was carried out on the Microsoft **Windows NT** platform. This illustrated the ease with which HERALD could be used to integrate third-party independent modules. The monitoring involved:

- *Terminal logging-on and logging out* – users taking part could install a small sensor application in their environment that would run when they log in and generate an event, and likewise generate a log-off event when it is terminated upon shell closure.
- *Application usage* – users use a toolbar to launch applications, and this generates events identifying what applications had been launched and when.
- *Document tracking* – clicking on a document in the **Windows**™ environment launches the application that is associated with that document for reading or editing. A document-listing application allowed document opening and generated events on these operations.
- *Telephony* – voice-enabled modems allow telephone calls to be made and received through a personal computer. This conveniently allows a wrapper around the **Windows**™ **Telephony** API to generate events about telephone activity and the numbers dialled. Furthermore, calls dialled or received (using caller-ID) can be matched to people or organisations through interaction with a user's address book.

- *Address-book, Email and Personal Organiser (PO) software* – comprehensive PO software suites like Microsoft **Outlook 98/2000™** integrate a user's calendar, task list, contact list, and also double as email interfaces and repositories. **Outlook** provides a very comprehensive programming model through which software can interface with most of its internal modules and monitor their activity. These wrappers were then made to integrate within the HERALD framework and issue events on email arrival, dispatch, entering of calendar appointments, appointment/meeting starting and ending, task creation and clearing.
- *Web browsing.* A HERALD aware web proxy was deployed and the user's web browsing software was set to divert all its HTTP retrieval operations to it. This proxy then handled web page retrieval transparently for the user's browser while generating events pertaining to the HTML pages being requested.

In this experiment, privacy was ensured by having all the events generated by a user's monitoring modules used only by other modules belonging to that user.

The above infrastructure enabled acquisition of sufficient information to be of use as an automated diary. Events from all the monitors were collected by a **User Monitoring Module**, which was also the client segment of an event repository. For one month, the storage required was of 4-8 megabytes for average use and for

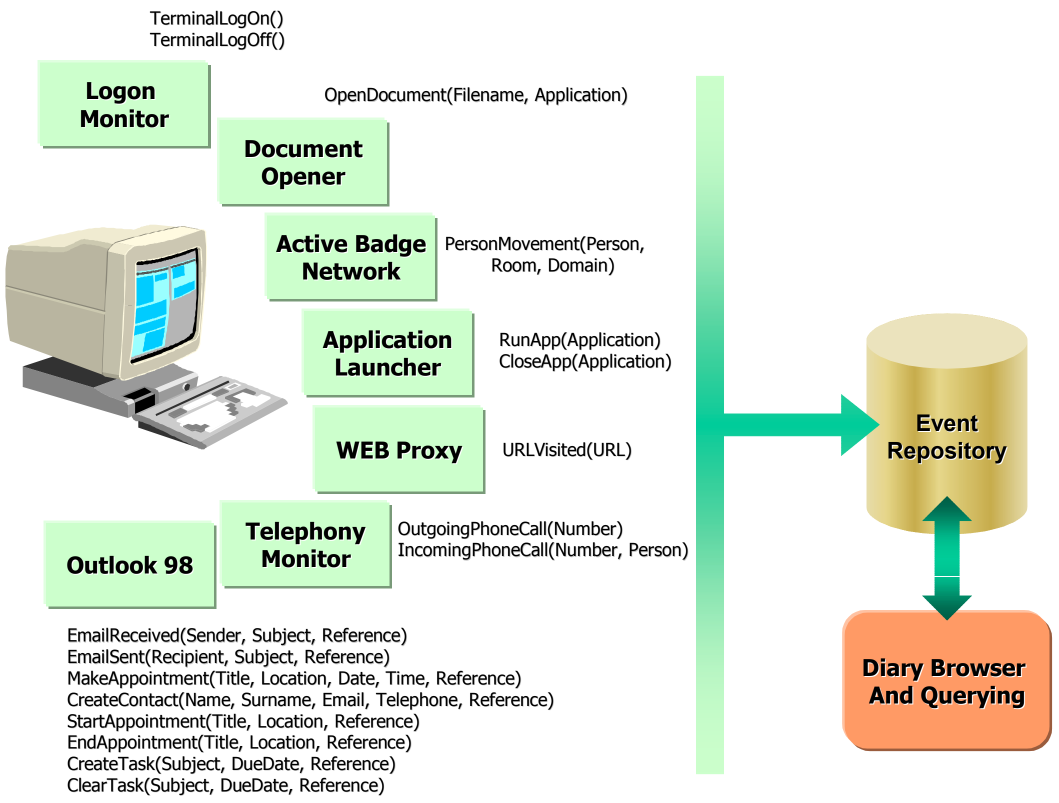
TerminalLogOn()
TerminalLogOff()
Logon Monitor
OpenDocument(Filename, Application)
Document Opener
Active Badge Network
PersonMovement(Person, Room, Domain)
Application Launcher
RunApp(Application)
CloseApp(Application)
WEB Proxy
URLVisited(URL)
Telephony Monitor
OutgoingPhoneCall(Number)
IncomingPhoneCall(Number, Person)
Outlook 98
EmailReceived(Sender, Subject, Reference)
EmailSent(Recipient, Subject, Reference)
MakeAppointment(Title, Location, Date, Time, Reference)
CreateContact(Name, Surname, Email, Telephone, Reference)
StartAppointment(Title, Location, Reference)
EndAppointment(Title, Location, Reference)
CreateTask(Subject, DueDate, Reference)
ClearTask(Subject, DueDate, Reference)
Event Repository
Diary Browser And Querying

**Figure 7.2**
Collecting and retaining information on a user's daily activities

uncompressed event data.

The user interface attached to the diary application enabled browsing of the raw data, and then its interpretation at various levels through application of TEQL queries. Upon entering a TEQL query the user would be presented with a refined view of the event data, this being the result of the query submitted. Output from multiple queries could be combined to create new data views, and these could be queried against in turn.

The following are examples of typical queries one could carry out from the data captured:

- *Show me the document that I edited after I had been to the meeting room where I met Jean (some day two weeks ago)*
- *Locate the details of a phone call I made 30 minutes after I received an email from George*
- *Locate the document I wrote after my appointment at the hairdressers some time last month and before I saw the Xerox PARC web page*
- *Show me the documents I edited on the 25th of this month from logging on until 12pm*

Although in this application, there is no notion of an integrated workflow from one component to another in a reactive manner, nevertheless it does demonstrate the ease with which independent applications can be turned into *event sources*. It is envisaged that it would be possible to use the event information collected to study users' interactions with each other and with electronic means of obtaining news (like web-sites and email).

The diary application was found to be useful as a tool for memory recollection, in particular for locating of documents. A future version detailing collaborative online sessions integrated with the above event information is being designed. This would then aid recollection of ideas and of concepts discussed in audio/video conferences. At present, this would only be available through keeping of video and audio logs (as in [LN93, LBC+94]) and manual annotation of the records.

## 7.4 Thin-client activity capture and replay

This thin-client session capture tool illustrated the feasibility of both temporal searching and replay for playback of computer sessions. This has potential for interface customisation and studying of human learning patterns.

Thin client platforms are becoming important tools in a number of environments like the active home, where ubiquitous terminals of low computing potential can be widely deployed. Likewise, the widespread use of handheld devices like personal organisers, palm computers and smart mobile phones, with severe power-consumption constraints (and thus performance restrictions), is bringing about a revival of client/server computing where as much computation and state information as possible is moved from the client to the server.

The **Virtual Network Computing** (**VNC**) technology from AT&T Laboratories Cambridge [RSWH98] pushes this thin-client computing model to an extreme by

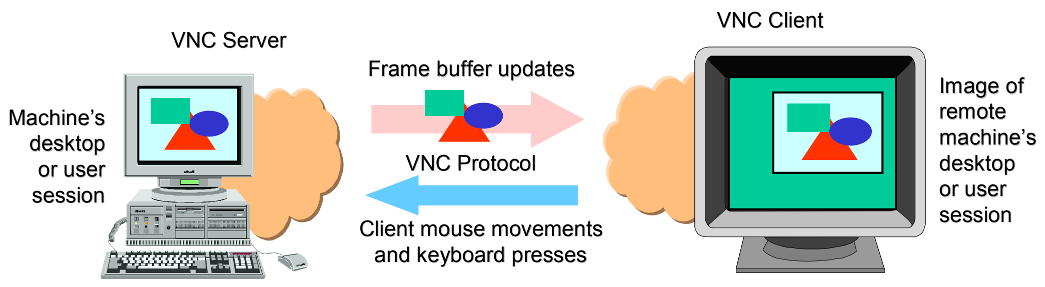
VNC Server
Machine's desktop or user session
Frame buffer updates
VNC Protocol
Client mouse movements and keyboard presses
VNC Client
Image of remote machine's desktop or user session

**Figure 7.3**
The **VNC** protocol

moving the execution of applications completely to the server, leaving the client stateless. Without changing any windowing system, **VNC** can run on any platform and breaks it into client/server pieces, namely the **VNC** server, the **VNC** client, and the **VNC** protocol that connects the previous two (Figure 7.3). The **VNC** server executes all the applications and generates the frame buffer. The **VNC** client displays the frame buffer and accepts user input.

The server component of **VNC** captures the desktop of the machine the server is hosted on. It then forwards this mirror of the desktop to the client component running elsewhere. The client displays this copy of the desktop through whatever display or windowing technology is available on the client platform, and allows the user on the client machine to interact with that desktop as if they were sitting at the console of that machine. Input devices like the mouse and keyboard can be controlled, and should another user use the console of the server machine, these would be shared with the remote user employing **VNC**. In this way, **VNC** enables a machine's desktop (or user session) to be used remotely in a way analogous to the **X** windowing protocol. The difference between **VNC** and **X** is that whereas the **X** protocol requires heavyweight clients to interpret drawing and display primitives, **VNC** carries out all computation at the server, and sends only frame-buffer updates and display segments to its lightweight client component. This has enabled **VNC** to be ported to several platforms and windowing environments, like **Windows**™, several **UNIX** flavours, and some hand-held operating systems. **VNC** also enables several users to simultaneously view the same desktop and is thus useful for classroom demonstrations. Inversely, an instructor or a researcher investigating an application's usability characteristics can monitor someone else using that application.

In this experiment, carried out in collaboration and documented in [LSBH00] by Li et al., the event repository developed in the course of this investigation was employed to capture and store the events sent from the **VNC** client to the server. HERALD's event transport functionality was not employed in this application, as **VNC** already uses a proprietary event protocol. The event repository was interfaced to this event mechanism and **VNC** through the writing of a **VNC** proxy module that was also the repository client component. This proxy server was seamlessly introduced between the **VNC** client and the server to intercept and manipulate the message streams in the **VNC** session [LSH99] (see illustration in Figure 7.4). Frame buffer update messages and various *event* messages that denote user actions (mouse clicks, window focus operations, entering of text) were retained. In effect, this is equivalent to capturing the user activity input on the **VNC** desktop, and enables playback of the whole or part of the session captured. Furthermore, the events serve as indices that can be queried to locate specific intervals within the user's activity. This capture and playback is useful for carrying out usability studies, as well as for the users themselves to review their activities. The resulting movie allows the reviewing user to view the captured activity with VCR-like control.

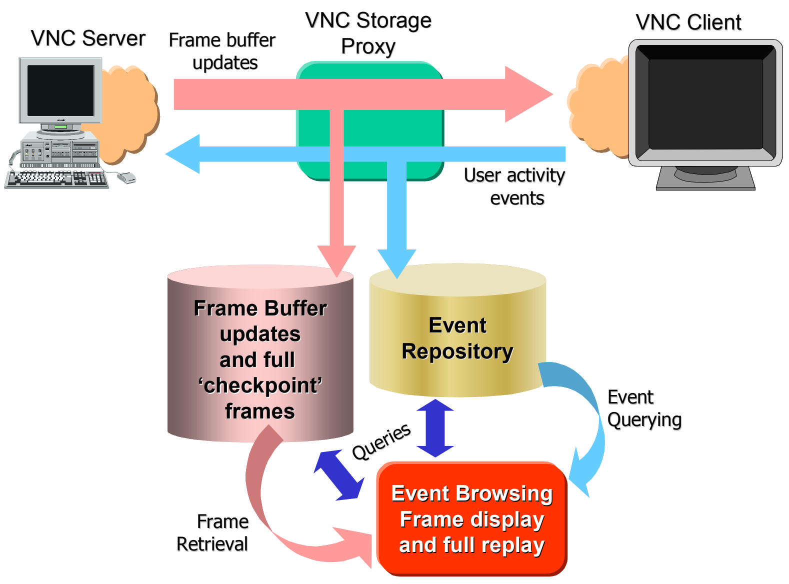

**Figure 7.4**
Inserting a storage proxy with an event repository to capture user activity

The events stored are used as indices into the frame-buffer updates. Firstly, there are *intentional annotation events*, which are indices that users create during a work session for marking particular time points or segments of activities or points of interest. These notes are time-stamped by the event repository and act as a temporal index into the recording. Several activities can be marked in this fashion, like text highlighting. Other automatically generated indices, termed *side-effect events*, refer to *key* (keyboard entry), *pointer* (mouse click) and *bell* (software generated notification) events. *Derived events* are produced by automated analysis of detailed multimedia records. For example, by analysing the frame buffer update messages, the proxy calculates the size of the frame buffer area that has changed since the last update and generates events pertaining to it.

In this application, the storage proxy presents a graphical user interface to the user. There is no requirement for the user to enter queries directly into any query language. A graphical menu-driven interface allows selection and customisation of a number of pre-set search, browse, retrieval and playback modes, and the proxy then translates these into TEQL commands transparently. The events retrieved from the event repository are then used to look up the appropriate frame buffer updates and reconstruct thumbnails for browsing or regenerate a session directly.

The retrieval queries available enable the user to locate (1) commands typed in at the console, (2) text entered into applications, as well as attempt to locate (3) any of the events captured by their type and by specifying templates for their parameters. Sequences of these events can be defined and searched for. When matches are located, the query interface locates and re-builds the display frames corresponding to when these events took place, and these are displayed in the form of thumbnails. The user can then choose to view a replay from that thumbnail onwards at the full original size, and at the same speed of occurrence, or faster.

This case study illustrates how the event repository can be employed with a proprietary event protocol. In measurements carried out in the context of this experiment to determine the latency of a retrieval operation, minimal performance latency was due to the event repository parsing queries. Most of the time was spent on retrieving the checkpoint image frames and then parsing all the incremental updates until a frame could be displayed. Increasing the frequency of 'checkpoint full frames' made this value tend towards a minimum figure of two seconds, a fraction of which is due to querying the repository.

## 7.5 Active Management of multi-service networks

This experiment illustrates the use of multiple event repositories working in collaboration. While live events enable service management and administration, analysis of event histories is employed to determine the effectiveness of management policies and to guide their evolution.

The increasing need for rapid introduction of new services and highly customised service offerings poses additional management challenges for today's networks. In order to address this, an investigation into management through event notification and storage was carried out in collaboration with BT Laboratories (Martlesham Heath, UK). The outcome of this investigation is described by Marshall et al. in [MBS+99]. Catering for modern customer service requirements requires the deployment of several specialised servers: examples being web servers, file servers, archive servers, e-commerce servers, mail servers, multimedia servers; working together to provide an integrated multi-service infrastructure. However, the reality of the marketplace often does not allow tight integration approaches to providing such a comprehensive solution. In order to be able to utilise third-party off-the-shelf products, loosely integrated event-driven approaches are feasible and practical. One powerful technique for providing such a service by a major information service provider that requires scalability is to group these services into clusters of servers, each running one or more services. In such a cluster, a number of machines (capable of being instructed to launch any service required) provide redundant backup in case of failure or overload, and management servers monitor and analyse the performance and load factors of the running servers. Gateway servers then act as bridges between these clusters, and not only propagate management information for

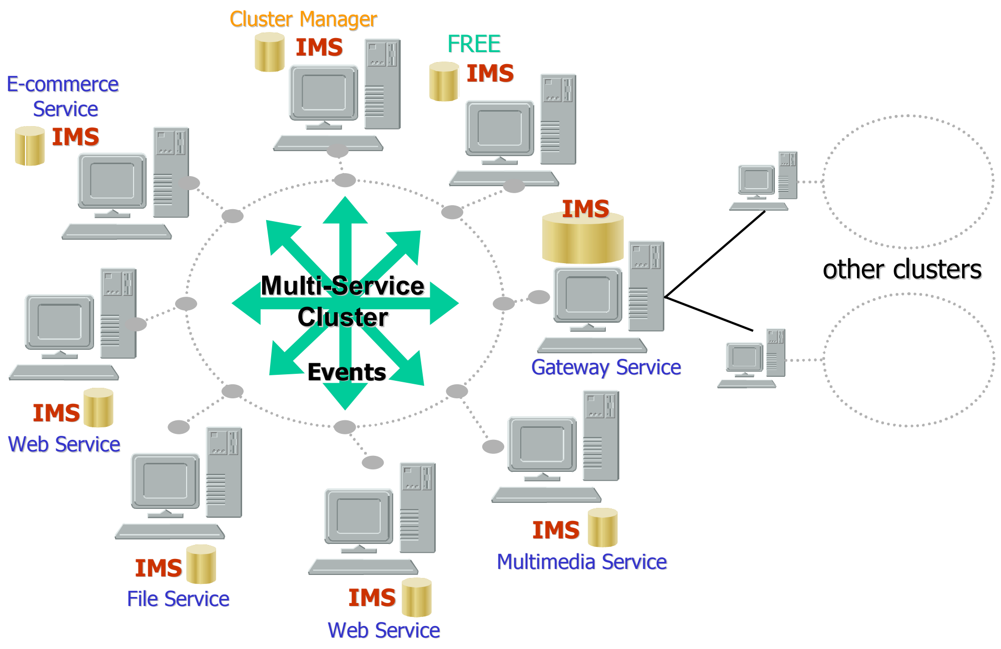

**Figure 7.5**
A multi-service cluster with a deployment of **Information Management Servers** with embedded event repositories

synchronisation, but also provide auditing and forwarding of client requests in case of cluster overload. This is shown in Figure 7.5.

The solution proposed as the outcome of this research employs *Active Management* (event-driven) based on role-driven policies and **Application Layer Active Networking** (**ALAN**) [FG98]. Information handling problems are avoided by using a lightweight scalable mechanism for information transfer. An **Information Management System** is deployed, which consists of a number of **Information Management Servers** (**IMS**'s), each of which encapsulates an event repository and captures management information. Events are propagated through HERALD's event transport. Thus, each specialised service, like the web server, is attached to an **IMS** (most likely running on the same machine), which can extract events from it as well as inject actions into the server according to events received. For example, an event generated from the web server indicates its current connection load. The **IMS** attached to the **Cluster Manager** service registers interest in this at a high delivery priority and can therefore monitor its load, starting another web server if need be. Similarly, it can send an event to the web server requesting it to stop handling client requests and shut down (as it might not be needed any longer, or another service is to be launched on that machine). Auditing servers also register interest in such events, but at the lowest priority.

The management system controls the services running within the cluster through an **IMS** at each network node, each of which is capable of autonomous actions. The actions to be undertaken at each entity are determined using knowledge that entity possesses about itself and policies (these being supplied by remote managers) that specify responses to system events. Typically, the local knowledge will consist of status data on elements controlled by the autonomous entity, and will constrain the number of policies that are applicable to an event instance. Each entity has a policy, defined by its administrator, which expresses the local precedence order of the management roles authorised to provide policies. All of the policies are based on the work of Sloman et. al. [LS97, Slo94] and allow manager rights (*authorisations*) and responsibilities (*obligations*) to be linked to the managers' role.

The actions possible include download and execution of management utilities and other executable programs that may be required but are not already installed. There are therefore policies (distributed with the programs) that specify the usage and behaviour of any programs added to the system using the active capability. The active capability is provided using the techniques discussed in [MCC+97]. Program policies are associated with the strongest role of the program provider at the entity where the program is used.

All **IMS**'s encapsulate event repositories that store and then periodically forward their data to other more centralised event repositories in the cluster. Therefore, two types of repositories are employed; lightweight partially functional servers which can be co-located with any entity, and fully functional servers which will typically be located at gateway nodes. This arrangement takes advantage of the flexible architecture employed in the event repository, which enables functional modules to be disabled; thus balancing the complexity of querying required against the computational power available at a node. While simple but fast event stores offer a load balancing and traffic controlling function, fully featured event repositories allow management information analysis.

Event repositories at most services are therefore cut-down in functionality to reduce their requirements footprint, with no temporal querying available. These stores retain their contents for a limited amount of time and space, and forward their contents to the primary event stores at the cluster gateway **IMS**'s when the time/space threshold is approached.

## 7.6 A 'mixed-reality' collaborative environment

The aim of this case-study application (named **CURVE** - Collaborative Unified Real and Virtual Environment) was to explore the potential of collaboration between real and virtual remote users in a common environment. As well as enabling review of the interaction of these real/virtual users, the event capture and storage service deployed also enabled replay of movement of the real users in situations like emergency evacuations. This application explored the feasibility of session playback to identify dangerous bottlenecks in buildings that could hamper rapid evacuation in case of an emergency.

In order to achieve this, a real working space (part of the Cambridge Computer Laboratory) was mirrored in virtual reality (VR), where the VR representation was as accurate as possible to the real world. This virtual reality model could be accessed

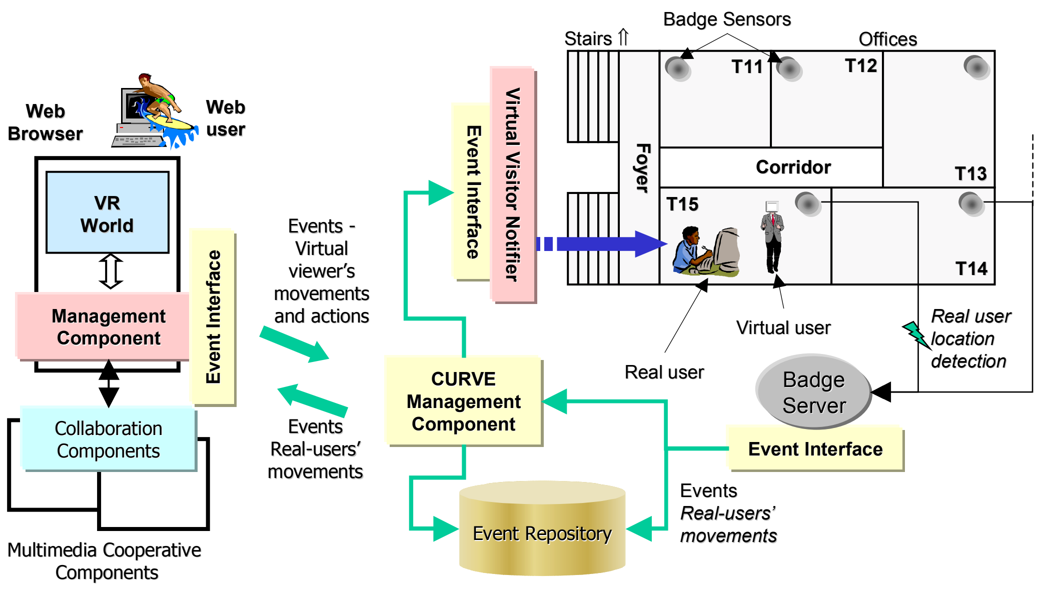

**Figure 7.6**
The **CURVE** architecture

through a web page, so it was available from anywhere. Although any user could download and navigate the virtual world, one was required to register and obtain a user identifier and password to be able to interact and collaborate with the 'real users'. The virtual reality world downloaded is dynamically modified as the user navigates through it to reflect activity in the real working environment. There are two aspects to this mirroring of a real environment into virtual-reality – *mirroring the building itself*, and *mirroring the real users that populate it and their activities*.

An accurate virtual reality model of part of the Computer Laboratory was built. The accuracy is reflected in the representation and dimensions of the building, floor layout, and rooms; right down to the level of the furnishing and textiles adorning each room. This enabled an individual familiar with the real building to immediately feel at home while navigating the virtual space, and thus be able to quickly proceed to the locations where (s)he wishes to visit or meet people. One of the problems associated with virtual-reality multi-user environments is the artificial nature of the worlds they present to users. In this application, this was not really an issue as the virtual world closely models an existing building. To further emphasise this, real-world movement is enforced, with active constraints like gravity. For example, to move up a floor, one has to take the stairs or use the lift. Since the virtual world created is so accurate in its representation of the real building, it has been used to visualise replay of users' movements in situations like emergency evacuations.

*Real users* were mirrored in the virtual world. When remote users visit a virtual office they will view avatars representing the real users 'really' present in the physical analogy of that office. Since recognizing someone from their avatar is important, several avatars are employed to represent people, and in addition, head photographs (mug shots) are used to customize each individual's avatar with their face. Knowledge of a real user's location and activities is achieved by using information gained from a number of sources. The **Active Badge** system [HH94] deployed throughout the real building enabled monitoring of the location of any individual who wishes it. Additional information acquired from software components running within an individual's desktop environment was employed (see Section 7.2). For example, if a user was currently logged in at a workstation with no idle time within that office, the system concludes that (s)he is using that machine. The avatar is then displayed seated or located next to the VR representation of that machine. If users move from one location to another, their movement is detected and fed back into the virtual world, which dynamically generates and displays their avatars moving in between the corresponding locations.

When virtual users navigated through the virtual world, their movements were captured and sent back to the Department, where they were then fed into real users' collaboration software. From the real users' perspective, they were made aware of their 'virtual visitors' through a software component running on the workstation nearest to their location (in the same room). This informed users whenever a virtual user was entering their office, and they could 'page' the virtual user. Virtual users can see other virtual users navigating through the virtual world, and can undertake interaction with them in a similar fashion to real-world users. Figure 7.6 illustrates the architecture behind **CURVE**.

**CURVE** is built around HERALD. An applet resident in the **CURVE** 'access web page' interacts with the third-party browsing software that displays the virtual reality world (defined in **VRML97** [ISO97]), and can dynamically modify it and its constituent objects. From this viewpoint, the application appears as in Figure 7.7. This applet also monitors activities in the virtual world, like users' movement and their interaction with the objects that make up the world; mapping these to HERALD events. It therefore acts as both a HERALD event source and client, and communicates with other HERALD components residing at the Computer Laboratory. The collaboration modules that can be launched to carry out audio-video collaboration between users are also active components. Classes of event include user interaction events, such as drawing lines, typing text, clicking on objects in a video clip, virtual reality viewpoint movement, and avatar selection.

**CURVE** supports the storage and retrieval of activities, enabling past sessions to be replayed at a later point, in whole or in part. It also allows the review of scenarios involving mobile users, for example emergency evacuation

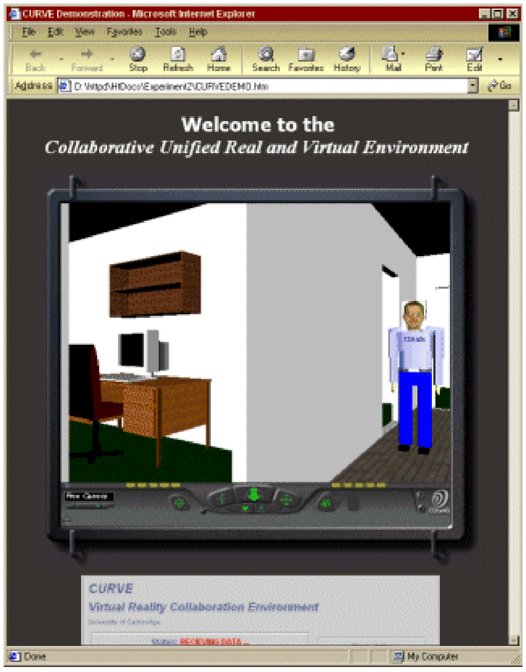
CURVE Demonstration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address D:\httpd\HtDocs\Experiment2\CURVEDEMO.htm
Go
Welcome to the
Collaborative Unified Real and Virtual Environment
CURVE
Virtual Reality Collaboration Environment
University of Cambridge
Status: RECEIVING DATA ...
Done
My Computer

**Figure 7.7**
The mobility visualiser.

procedures. In this experiment, users' movements during a fire drill were monitored and captured by the event repository attached to the **Active Badge Event Source**. The event stream captured was then fed back into **CURVE**, which used it, just as it would with live data, to visualise individuals' progress through and out of the building. This enabled identifying of problem spots within the building, such as bottlenecks affecting the safe exit of a large number of people. This can be a valuable tool for an architect to identify design flaws in a building, and assist with making modifications or providing alternative exits. In the evacuation scenario, **CURVE** was not employed as a collaborative environment as such, but its ability to mirror a real world and its users was used to study that real world during an interval of interesting activity. The advantage of using a virtual reality mirror over closed-circuit camera footage is that in the virtual world one can change one's viewpoint continuously as opposed to being restrained to viewing from fixed and specific viewpoints.

In summary, events describing real and virtual users' movements and activities were captured over time and stored in an event repository. A replay could then be initiated from the repository, feeding back the event streams denoting people's movements within the building into the virtual reality mirror of it. This could be done at the same speed as the original time scale, or in faster multiples. The replay could be made from the beginning of the capture session, or from *any temporal point to any other point*. TEQL replay queries could be made interactively into the replay module, which would then return a set of replay intervals that match the query provided. The user can then select which replay to carry out, this then being fed into the virtual reality world.

## 7.7 Summary

The aim of these case-studies was to explore the potential of event storage in various application domains having different integration, storage, and retrieval requirements.

The diary application illustrates the ease with which information can be gleaned from several sources and integrated using the event notification approach. The approach taken in this experiment enabled locating information based on its temporal context by applying the principle of human memory. By cross-examining the event histories pertaining to different users and relating them, this tool could be the underpinning of an analysis of workflow patterns in the workplace.

The **VNC** thin-client session capture tool allows applications to be evaluated for the usability of their interfaces and feature-set, as well as assist in teaching within classroom environments. This illustrated the feasibility of both temporal searching and replay for playback of computer sessions. Coupling this with post-session high-level semantic analysis and a knowledge base (such as that prototyped in **Lumiere** [HBHHR98]) has potential for interface customisation and studying of human learning patterns.

The multi-service network management infrastructure introduced in Section 7.5 addresses the complex issue of managing the various servers that a modern information provider needs to service scalable customer requirements. The solution illustrates the use of multiple event repositories working in collaboration. Therefore, while live events enable service management and administration, analysis of event histories is employed to determine the effectiveness of management policies and to guide their evolution.

Finally, the **CURVE** tool allows remote users to visit a virtual replica of a real building, and view and interact with the real users working and populating it. As well as enabling review of the interaction of these real/virtual users, the event capture and storage service deployed also enables replay of movement of the real users in situations like emergency evacuations. This application explored the feasibility of session playback to identify dangerous bottlenecks in buildings that could hamper rapid evacuation in case of an emergency.

Chapter 8

# Analysis

In order to evaluate the solutions presented in this dissertation, it is useful to recall the original research aims as documented in Chapter 1. The proposed event storage solution required provision of:

- *a model for representing generic event instances,*
- *an architecture for seamless integration of the functionality and services of an event storage repository with different environments,*
- *a flexible storage service that embeds a high-performance storage paradigm tailored for the particular nature of event data,*
- *a powerful interface for retrieval and replay of information from event stores.*

Chapter 2 then highlighted a number of application domains where event notification is recognised as a feasible and powerful communications model, but where further advances require review and analysis of event histories. These research issues were therefore addressed with an emphasis towards generic applicability. This approach can be unsafe as it can lead to elegant solutions that in practice cannot meet the bespoke requirements of any particular application. In order to avoid this pitfall, flexibility and customisation was built into the designs themselves. The idea behind this is that application developers can adopt the above services in order to built their application while tailoring them to their particular requirements and constraints.

This chapter provides a synopsis of the features of the service presented and discusses their merits and shortfalls. Where appropriate, it highlights directions for future research.

## 8.1 A model for representing generic event instances

The event model presented in this dissertation (Chapter 4) offers a generic way of representing and reasoning about events. In past, event notification systems have been hampered by providing unstructured event representations that severely limited their application outside a few bespoke applications. The model proposed avoids any application-oriented emphasis in order to avoid this fate. Not only is it straightforward to map events from all sorts of hardware and software devices to the event model proposed, but it also becomes possible to use the same event data generated within one specialised application within another of completely different scope.

The event model defines a type system for events where event instances belong to a structured type, whose attributes are of any of the types defined in the Object Data Management Group's data types. These strict definitions on the representation, size and range of types like integers, real values and strings, ensure that the event type system is not tied down to any particular programming language data model. At the same time, the ODMG data model provides bindings to the type systems of the most popular languages, like **Java™**, **C/C++** and **Smalltalk**, and is very close to the type system defined by the Object Management Group, the other consortium relevant for distributed systems design. These factors contribute towards rapid integration, processing and usage of events within applications across heterogeneous platforms.

Since events are typed and structured, services can determine precisely what event information they require and specify fine-granularity filtering, providing for highly client-focused event notification, with reduced network load and client-side computation required.

The future of distributed systems lies in dynamic open infrastructures where application components can be written by different organisations, and can dynamically join or leave a distributed computation. Components can come and go, provide services that others can make use of, and scour a system in order to discover information that they can consume. This scenario precludes tightly coupled approaches, where event schema are statically propagated using compiled skeleton files or stubs.

In this model, event instances only contain a reference to their type and their attributes' values. It is therefore proposed that event schema, i.e. the structures that define what is meant by an event type and which collectively make up the event taxonomy available to an application, are defined at the event sources where the events are generated, or at local event brokers that have knowledge of the events available in their domain. From here, they can then be propagated to entities wishing to consume events of those schema's types. Self-describing messages in formats like **XML** are useful for when an event consumer is interacting with message-based channels delivering events of different kinds, and is undecided as to what is useful to it. This is very expensive in terms of network bandwidth, due to the large message size, and in terms of the filtering and parsing required at the event consumer. Searching for and obtaining a descriptive scheme once seems to be a more practical solution, and is particularly effective when coupled with a federated network of event brokers [BBMS98].

Since event types can inherit properties from other events within an inheritance tree, applications can structure services around meaningful organisations of event types. In this way, registration and querying of event data can be carried out on base types that implicitly encompass event instances from all the sub-types.

In order to further facilitate discovery of information, scheme definitions enclose textual fields that describe an event type's purpose, and provide a set of keywords that can be used for classification and indexing. Likewise, within an event type's scheme, each attribute is not only named and typed, but is also tagged with textual descriptions of the nature of that attribute. Together, these allow for search services that locate useful information, and support automatic translation of event taxonomies in between domains. In addition, since schema are versioned, they can evolve over time without requiring re-compilation of components. Event services can serve events in the new and old format alongside each other without causing confusion.

Finally both *regular*, as well as *compound*, composite events are implicitly supported. Registering in a composite event is analogous to registering in a pattern of occurrence, and enables applications to determine the granularity of events they wish to see, without losing the ability to see the constituent primitive events if desired.

These capabilities go beyond the provisions of most event and messaging models, and lead one to conclude that the event model presented can more than meet the requirements of the application domains listed in Chapter 2. In Chapter 7, the diary application, the multi-service network management infrastructure, and the **CURVE** virtual reality system, all endorsed the event model presented. It adequately fulfilled their requirements. Furthermore, the model specifies that its structures themselves all be versioned, ensuring that the core model itself can evolve in response to emerging requirements.

## 8.2 An infrastructure for event notification and storage

The HERALD event notification and storage infrastructure (Chapter 4) provides a flexible framework for building distributed systems from independent components. HERALD provides the transport within which to implement and propagate structured events as defined by the above event model. By not imposing any system-wide constraints on data propagation and communication, HERALD embraces the loosely coupled model of interaction that, as argued in Section 8.1, represents the future of distributed systems and Internet-based software.

HERALD's underlying philosophy is that by wrapping a small layer of code around a device or software component, that component can be turned into an *active* component. This is a component that can: (1) either generate events that are of interest to other components, or (2) can register interest and consume events from other components, or (3) can have actions injected into it. In the third case, an application-specific federator module can be written that employs declarative rules to drive injection of actions (triggering of activity) inside one or more non-active components. This approach enables distributed systems to be rapidly and dynamically composed from several applications that were not intended for such integration. Section 7.3 demonstrates how this capability can be deployed.

The HERALD transport is characterised by its building block approach to providing specialised event services within an application. *Client modules, Source modules, federator modules, event repositories, brokers, composite event engines* and *action injection interfaces* can be seamlessly brought together by an application writer to provide a variety of specialised *mediator* components. Throughout, a comprehensive event registration service with a variety of registration policies provides for propagation of event information as defined by the event model described.

The registration policies supported by HERALD event sources provide clients with a means to apply several constraints on how events are notified to them. While this feature-set enables an event source to provide a comprehensive service to its clients, an application writer can choose to disable several aspects of it. This allows event sources to be used on platforms where memory, disk, and processor cycles are at a premium. The reflective nature of the interface that ties together clients and sources ensures that components can accommodate discovery of service availability and dynamic change.

Since both event sources and clients can embed event stores, event histories can be collected at various points within a distributed system. By directly enabling components to query each other with regards to their event histories, and to request retrieval of past events and event replays, HERALD provides the same level of access to event histories as to live notification. This is a novel contribution for a generic event transport.

## 8.3 An interface for retrieval and replay of event information

An event history represents many things in many applications. It can represent a history of interaction between cooperative applications, a trace of low-level system events, a sequence of stock prices, a listing of telephone calls from a telephone exchange, or a detailed record of a user's interaction with other users and with his/her digital environment. Chapter 2 gave further examples that indicate the breadth of domains where a history has uses. Each of these scenarios supports a number of applications, and within each, there are different requirements as to how an event history might need to be queried and analysed.

What underlies these applications' query requirements, however, is the singular nature of the data they need to process. Event data differs from other datasets in that it is ordered primarily by time, and analysis of it tends to revolve around looking at sequences and content over time rather than over relations.

Designing an interface language that addresses these diverse requirements is a challenge that is hard to meet, and this investigation has attempted to provide one that makes expressing such querying more straightforward that with conventional query languages. TEQL implicitly supports the notion that an event history is a sequence of data that is ordered by time and is likely to be queried primarily according to this property. It also provides the groundwork over which advanced event history analysis can be defined.

The important features of TEQL are as follows:

- *Programming flexibility,*
  Queries can be embedded through command requests that can be programmed within applications of the repository. While a query language provides more querying power than can be provided through a fixed application-programming-interface, the ability to embed queries within program code increases the range of uses of the interface.
- *Intuitive contructs,*
  The new constructs provided have been carefully selected to reflect their meaning. Although this does not come close to the clarity of a natural language, it allows one to express queries that would have been cryptic at best if expressed in conventional database query languages.
- *Conventional querying capability,*
  Specialised query languages frequently suffer from being very good at addressing niche queries but unsuitable at general-purpose conventional querying. By defining TEQL as a superset of **OQL**, event data can be accessed as with any ODMG-compliant database management system. In addition, new users of the language can rapidly learn to use it as it only requires a minor shift of thinking about the data to devise queries.
- *Type-independence and semantic context,*
  TEQL allows queries to be defined in temporal entities rather than specific relational structures or typed objects. This implicitly abstracts away the type of an event, and allows a user to reason over the entire range of events within an event history. Sessions enable a user to apply a context around related events, and derived sessions allow one to selectively generate views of the event data in terms of their semantics rather than their structure.
- *Composition of events and derivation of abstract representation,*
  Multiple instances of patterns of sequential, although not necessarily contiguous, event occurrences can be located, and turned into composite events. This pattern recognition function allows one to reduce the granularity of the event information. In order to represent activities that last for an amount of time, intervals can be extracted from these composite events, as well as from primitive events and from real time values. The result sets of several queries, that may contain selected primitive events, composite events and intervals, can be joined to create a new derived view of the original event history that reflects some application-specific higher-level interpretation. These derived views can be made persistent, queried on themselves, and propagated to remote event repositories.

- *A formalism for temporal entities,*
  A temporal formalism is defined that clearly lays out the nature of events, timepoints, intervals, and timelines. It also provides meaningful definitions for ordering between all these temporal entities.
- *Temporal relationships between events, real-time, date and calendar entities.*
  Finally, TEQL directly endorses the model of real time as the desirable representation of global time. This controversial approach is required within emerging loosely coupled architectures of dynamic nature, and is justified by a number of studies [Die96]. This is reflected in the fact that TEQL maps timestamps to values of real time, and provides constructs that enable users to directly relate events to relative intervals of real time, and with respect to real time absolute values like time, dates, and calendar entities.

The retrieval performance of the prototype implementation of TEQL has not been empirically evaluated since a study into algorithms for efficient temporal query evaluation and optimisation was beyond the scope of this document. This is a strong line for further research. Several aspects of the language indicate that there is scope for optimisation of query evaluation, and any thorough treatment might suggest a revised storage and indexing structure. It is an open issue how to balance this retrieval performance against rapid storage. Using optional off-line generated indices or persistent views [JMS95] within certain repositories appears to be a promising approach. Seshadri et al. [SLR94] suggest ways of optimising queries over ordered sequences that are similar to temporally ordered event data. Although they assume that a full sequence is available at all times, their algorithms are open to modification for support segments of complete event data instead.

TEQL also allows partial match queries to be carried out through its template matching constructs. Partial matches are difficult to optimise in any database, particularly because even if an index were available for each event attribute, access would still be far from optimal. Current approaches suggest that multi-dimensional index structures, like the R-tree [Gut84] and the more optimal R*-tree [BKSS90], yield the best performance for access of this kind. However, whether creating an R-tree for each event type is the best approach for optimised template matching within the event repository is a research issue. In particular, what might reduce the effectiveness of such an index is the fact that queries in TEQL are usually restricted to a sequence of events rather than across the whole database. Although this supports partitioning of the data into smaller more manageable chunks for query evaluation, it can render an expensive global indexing effort redundant. Therefore, an R-tree that indexes the whole data set might not be that useful. This suggests that arbitrarily diving the event data into temporal segments, and providing an R-tree for each segment might be a better compromise. Queries that range over more than one temporal segment would then have to access all the R-trees of the segments involved and merge the result-set.

TEQL could be further enhanced to allow users to define their own calendar with its calendar entities (*I-Times*). This would require a formalism for specifying I-Times such as that proposed by [Ter97].

In conclusion, an exciting area for future research consists of investigating the possibility that an AI-derived language could be mapped onto TEQL or a revised version of it. These languages come much closer to natural language expressiveness that any database query language available.

## 8.4 A storage service

The structured design proposed for an event repository balances the desire to provide a generic storage solution against the unavoidable breadth of integration requirements presented by applications.

Applications that require the use of an event store bring with them constraints on storage and processing power. They also differ in how they are to interface with the event repository, and often require dedicated services that are particular to their function and usage of event histories. For this reason, the repository architecture is functionally divided into what are termed into a repository server component and a repository client component. While the server component of the architecture is kept as generic as possible to provide a core set of functionality, repository clients can be tailored to be highly specific to an application. The actual storage and retrieval functionality of the event repository is supplied by the repository server.

A number of repository clients can interface with a repository server. The latter is composed of a number of modules that provide storage and retrieval services, and some of them can be disabled if required to reduce resource requirements. Likewise, the client-side component libraries provide for customisable features like timestamp mapping, data model translation, and sliding event observation windows that can be enabled only if required. While any customisation of the server reflects on all its applications, client specialisation applies only for its application.

The custom storage subsystem makes away with unnecessary conventional database overhead and applies techniques developed in database transaction log keeping and event tracing within distributed debugging systems. The log-based solution provides for very fast storage of incoming event streams, and organises event data by its type and temporal order. The only performance compromise made is in real-time generation of session indices and duplexing of log pages. An index of the temporal intervals represented by log pages is also retained by the **Log Manager**, aiding retrieval queries based on temporal separations and values of real time.

In order to evaluate the raw writing performance of the prototype log-based storage system a number of trials were run. The test environment used consisted of a repository client and repository server co-located on one machine, with the client generating events and depositing them into the store. The machine was an Intel Pentium-II 450Mhz machine with 256Mb RAM, 14GB of EIDE (Mode 4) hard-drive storage running **Redhat Linux 5.2**. Events were deposited both through TEQL queries and directly through the stream connection. Ten thousand events were deposited by the client, and the total time this operation took was used to work out the average time required to deposit an event. The result is shown in Table 8.1.

| | |
|---|---|
| Time to parse deposit query | $100\mu s$ |
| Average time to insert 20byte event | $264\mu s$ |
| Average time to insert 1K event | $600\mu s$ |

**Table 8.1**
Performance figures for the Event Repository

| | |
|---|---|
| Average time to insert 20byte event | $35ms$ |
| Average time to insert 1K event | $45ms$ |

**Table 8.2**
Performance figures for Microsoft **Access 97**

For the purpose of a comparative evaluation, the same data event data was also stored into a popular and widely deployed database, Microsoft **Access 97**. **Access** was running on a **Windows NT** machine of identical specification. The event data was submitted through the database's **ODBC** access interface. These results are shown in Table 8.2.

Packaging a query, passing it over a local socket, parsing it, and returning an acknowledgement, adds around $100\mu s$ to the deposit of each event.

This overhead is reduced when events are streamed in directly, since it need only be set up once at the beginning of the event input streaming. In this case, the above results indicate that on this platform the repository could capture around 4000 events of size 20bytes (this being normal size for most events) per second before the storage thread starts running out of bandwidth. It was not possible to isolate the effects of any write-caching that might have been applied by the operating system or disk controller, so this result might be artificially good, and might not be applicable over very long periods of time.

During this measurement, incoming event buffering and queuing were disabled. With this enabled as per normal operation, the queue would be increased dynamically to buffer events should disk bandwidth starvation occur. This illustrates that the log-based storage structure delivers the crucial high-speed event deposit performance. Furthermore, it is vastly more efficient at this than the **Access** database.

Finally, the **History Propagation Engine** provides an application with several options on how to capture and retain its event histories. It can choose to reduce network bandwidth by retaining events locally, and only forwarding histories at periodic intervals. This also enables one to address the issue of unsynchronised network clocks by merging local event traces at one centralised location in order to provide a globally ordered history. Application components can also utilise cut-down event repositories on platforms with scarce resources since they can rely on later being able to propagate the captured histories to a more potent remote event repository.

In conclusion, the generic repository architecture addresses the main issues of flexibility in the face of different application requirements. The prototype implementation indicates that the proposed design delivers appropriate event storage performance.

# Chapter 9

# Conclusion

The future of distributed systems lies in dynamic open infrastructures where applications are composed of loosely coupled independent components. These dynamically join or leave a distributed computation, and dynamically discover information about each other and each other's capabilities. This dissertation has argued that the lack of a generic model for event representation and notification has restricted the development of these systems by restricting interoperability and scalability. Furthermore, in order to empower existing applications and enable novel solutions, a crucial service within event-driven systems is capture, persistent storage, and meaningful retrieval of the event information driving these systems.

This dissertation has addressed these issues by contributing:

- a generic and flexible model for representing event types, that allows reasoning on generic events without being restricted to the scope of the application that generated or used the events,
- an infrastructure that provides event notification and capture of events at various conceptual locations within a distributed application,
- an event storage architecture, which can be customised to meet the individual performance and functional criteria of different applications,
- an interface for retrieval and replay of event information that supports the notion that event information is temporally ordered, and provides for query constructs that emphasise this property.

The design proposed was verified through a working implementation and deployed in a number of application scenarios. These novel applications required state-of-the-art support that cannot be provided by any other middleware infrastructure.

# Bibliography

[**4Ti99**] 4Tier Software. OpenMOM. http://www.4tier.com, 1999.

[**AAEM97**] G. Alonso, D. Agrawal, A. El-Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *IEEE Expert*, 12(5), 1997.

[**Abu99**] Abuzz. The Beehive System. http://www.abuzz.com, 1999.

[**AL95**] N. Ackroyd and R. Lorimer. *Global Navigation: A GPS User's Guide*. Lloyds' of London, 2nd edition, 1995.

[**All83**] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832-843, November 1983.

[**All91**] James F. Allen. Time and time again: The many ways to represent time. *International Journal Intelligent Systems*, 6(4):341-355, July 1991.

[**ATT99**] AT&T Research Laboratories Cambridge. The smart beverage dispenser. http://www.uk.research.att.com/cgi-bin/coffee, 1999.

[**Bat95**] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM Transactions on Computer Systems*, 13(1):1-31, February 1995.

[**BBHM95**] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using events to build distributed applications. In *Proceedings of the 2nd International Workshop on Services in Distributed and Network Environments*, pages 148-155, Whistler, British Columbia, USA, 1995.

[**BBMS98**] John Bates, Jean Bacon, Ken Moody, and Mark D. Spiteri. Using events for the scalable federation of heterogeneous components. In *Proceedings of ACM SIGOPS European Workshop 1998*, pages 58-65, Sintra, Portugal, September 1998.

[**BCTW96**] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378-421, October 1996.

[**BEM94**] N. Belkhatir, J. Estublier, and W. Melo. *ADELE-TEMPO: An Environment to Support Process Modelling and Enaction*. Advanced Software Development Series. John Wiley and Sons, 1994.

[**Ben99**] Richard Bentley. Awareness in work places. Personal communication, 1999.

[**BHB96**] John Bates, David Halls, and Jean Bacon. A framework to support mobile users of multimedia applications. *Baltzer Mobile Networks and Nomadic Applications*, 1996.

[**BHI93**] Sara A. Bly, Steve R. Harrison, and Susan Irwin. Media Spaces: Bringing people together in a video, audio, and computing environment. *Communications of the ACM*, 35(1):28-47, January 1993.

[**BHS93**] Albert N. Badre, Scott E. Hudson, and Paulo J. Santos. An environment to support user interface evaluation using synchronized video and event trace recording. Technical Report GIT-GVU-93-16, Graphics, Visualization and Usability Center, Georgia Institute of Technology, USA, 1993.

[**BK95**] I. Ben-Shaul and G.E. Kaiser. *A Paradigm for Decentralised Process Modeling.* Kluwer Academic, 1995.

[**BKSS90**] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* , pages 322–331, 1990.

[**BPS99**] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical report, World-Wide-Web Consortium, February 1999.
http://www.w3.org/TR/REC-xml.

[**BRS+94**] R. Bentley, T. Rodden, T. Sawyer, J. Sommerville, I. Hughes, D. Randall, and D. Shapiro. Ethnographically informed systems design for air traffic control. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 287-298, 1994.

[**BSHB98**] John Bates, Mark D. Spiteri, David Halls, and Jean Bacon. Integrating real-world and computer-supported collaboration in the presence of mobility. In *Proceedings of IEEE seventh International Workshops in Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 1998)*, pages 256-261, Stanford, USA, 1998.

[**CB97**] R.G.G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[**CDF98**] G. Cugola, E. Di Nitto, and A. Fugetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 1998 International Conference on Software Engineering (ICSE '98)*, pages 261-270, 1998.

[**CDRW98**] Antonio Carzaniga, Elisabetta Di Nitto, David S. Rosenblum, and Alexander L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of the Third International Workshop on Software Architecture (ISAW '98*, pages 17-20, 1998.

[**CFS+94**] Earl Craighill, Martin Fong, Keith Skinner, Ruth Lang, and Kathryn Gruenefeldt. SCOOT: An object-oriented toolkit for multimedia collaboration. In *Proceedings of the 2nd ACM Conference on Multimedia*, pages 41-48, 1994.

[**CH94**] Gil Cruz and Ralph Hill. Capturing and playing multimedia events with STREAMS. In *Proceedings of the 2nd ACM Conference on Multimedia*, pages 193-200, San Francisco, CA, USA, 1994.

[**CHRW98**] A. Cichocki, A. Helal, M. Rusinkiewicz, and D. Woelk. *Workflow and Process Automation: Concepts and Technology*. Kluwer Academic Publishers, 1998.

[**CJ98**] C. Ma and J. Bacon. COBEA: A CORBA-based event architecture. In *Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems*, pages 117-131, June 1998.

[**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 VLDB Conference*, pages 606-617, Santiago, Chile, 1994.

[**CLFS93**] Earl Craighill, Ruth Lang, Martin Fong, and Keith Skinner. CECED: A system for informal multimedia collaboration. In *Proceedings of the 1st ACM Conference on Multimedia*, pages 437-444, 1993.

[**CM93**] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, University of Florida, USA, 1993.

[**CW82**] R.S. Curtis and L.D. Wittie. BugNet: A debugging systems for parallel programming environments. In *Proceedings of 3rd International Conference on Distributed Computing Systems*, pages 394-399, Miami, FL USA, October 1982.

[**DA96a**] Alan Dix and G. Abowd. Delays and temporal incoherence due to mediated status-status mappings. *SIGCHI Bulletin*, 2(28):47-49, 1996.

[**DA96b**] Alan Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 6(11):334-346, 1996.

[**Day88**] U. Dayal. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51-70, March 1988.

[**DBM98**] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In *Proceedings of the 2nd Intl. Workshop on Object-Oriented Database Systems*, Lecture Notes in Computer Science 334. Springer, 1998.

[**DFAB98**] Alan Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice-Hall, 2 edition, 1998.

[**DFWB98**] Nigel Davies, Adrian Friday, Stephen P. Wade, and Gordon S. Blair. An asynchronous distributed systems platform for heterogeneous environments. In *Proceedings of the eighth ACM SIGOPS European Workshop*, pages 66-73, Sintra, Portugal, 1998.

[**Die96**] Margaret A. Dietz. *Gathering and Using Time Measurements in Distributed Systems*. PhD thesis, Department of Computer Science, Duke University, Durham, NC USA, 1996.

[**Dut99**] Amitava Dutta-Roy. Networks for homes. *IEEE Spectrum*, 36(12):26-33, December 1999.

[**EE99**] Guy Eddon and Henry Eddon. *Inside COM+ Base Services*. Microsoft Press, September 1999.

[**FBSC93**] L. E. Fahlen, C.G. Brown, O. Stahl and C. Carlsson. A space based model for user interaction in shared synthetic environments. In *Proceedings of the ACM Conference on Human Factors in Computing*, Amsterdam, The Netherlands, April 1993.

[**FG98**] M. Fry and A. Ghosh. Application Layer Active Networking (ALAN). In *Proceedings of the 4th International Workshop on High Performance Protocol Architectures (HIPPARCH'98)*, June 1998.

[**Fid91**] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28-33, 1991.

[**FM82**] M.J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principle of Database Systems*, pages 70-75, 1982.

[**FMK$^+$99**] Geraldine Fitzpatrick, Tim Mansfield, Simon Kaplan, David Arnold, Ted Phelps, and Bill Segall. Instrumenting and augmenting the workaday world with a generic notification service called Elvin. In *Proceedings of the 6th European Conference on Computer Supported Cooperative Work (ECSCW'99)*, page 431, Copenhagen, Denmark, September 1999.

[**For91**] Ray Ford. Non-intrusive real time event capture. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 225-227, Santa Cruz, CA, USA, May 1991.

[**Fro95**] D. Frolich. *Mobile Personal Communications and Co-operative Working*, Chapter: Requirements for interpersonal information management. Unicom Seminars, 1995.

[**Gav92**] William W. Gaver. The affordances of Media Spaces for collaboration. In *Proceedings of the ACM 1992 conference on Computer supported cooperative work Conference CSCW'92*, pages 17-23, Toronto, Canada, November 1992.

[**GD93**] S. Gatziu and K.R. Dittrich. Events in an active object-oriented database system. In *Proceedings of the 1st International Workshop on Rules in Database Systems*, Edinburgh, UK, August 1993.

[**Gel85**] David Gelernter. Generative communication in Linda. *TOPLAS*, 7(1):80-112, January 1985.

[**Ger90**] C. Gerety. HP SoftBench: A new generation of software development tools. *Hewlett-Packard Journal*, 3(41):48-59, June 1990.

[**GHS95**] D. Georgakopoulos, M. Homick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2), 1995.

[**GJS92**] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th VLDB Conference*, 1992.

[**GKBF98**] S. Gatziu, A. Koschel, G.V. Buetzingsloewen, and H. Fritschi. Unbundling Active functionality. *ACM SIGMOD Record*, 27(1):35-40, March 1998.

[**GKP98**] R.E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the READY notification system. In *ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, September 1998.

**[GM95]** Ashish Gupta and Indepal Singh Mumick. Maintenance of materialised views: Problems, techniques, and applications. *Data Engineering Bulletin*, June 1995.

**[GN91]** D. Garlan and D. Notkin. *VDM Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, Chapter: Formalising design spaces: Implicit invocation mechanism, pages 31-44. 1991.

**[Gra93]** Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73-169, 1993.

**[GRWB92]** Saul Greenberg, Mark Roseman, Dave Webster, and Ralph Bohnet. Human and technical factors of distributed group drawing tools. *Interacting with Computers*, 4(3):364-392, 1992.

**[GT98]** A. Geppert and D. Tombros. Event-based distributed workflow execution with EVE. In *Proceedings of IFIP International Conference on Distributed Platforms and Open Distributed Systems (Middleware 1998)*, Lancaster, UK, September 1998.

**[Gut84]** A. Guttman. R-trees, a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD, International Conference on Management of Data*, pages 47-57, 1984.

**[Han88]** Jeffrey P. Hansen. Trend analysis and modeling of uni-multi-processor event logs. Master's thesis, Carnegie Mellon University, 1988.

**[Hay96]** Richard Hayton. *OASIS: An Open Architecture for Secure Interworking Services.* PhD thesis, University of Cambridge, March 1996.

**[HBH$^+$98]** E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, July 1998.

**[HFK95]** Nael Hirzalla, Ben Falchuk, and Ahmed Karmouch. A temporal model for interactive multimedia scenarios. *IEEE Multimedia*, (3):24-31, Fall 1995.

**[HH94]** Andy Harter and Andy Hopper. A distributed location system for the Active office. *IEEE Networking*, 8(1), January 1994.

**[HHH92]** M.L. Hammontree, J. Hendrickson, and B.W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proceedings of CHI'92*, 1992.

**[HI91]** C. Hewitt and J. Inman. Dai betwixt and between: From intelligent agents to open systems science. *IEEE Transactions Syst. Man Cybernet*, 21(6):1409-1419, 1991.

**[HS92]** J.P. Hansen and D.F. Siewiorek. Models for time coalescence in event logs. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS '92)*, pages 221-229, Boston, MA, USA, July 1992.

**[IBM99]** IBM Corporation. MQSeries. http://www.ibm.com/software/ts/mqseries/, 1999.

**[IET99]** Internet Engineering Task Force - IETF. Mobile IP Internet drafts. http://www.ietf.org/ids.by.wg/mobileip.html, 1999.

**[IKG92]** Hiroshi Ishii, Minoru Kobayashi, and Jonathan Grudin. Integration of interpersonal space and shared workspace: ClearBoard design and experiments. In *CSCW '92, Conference proceedings on Computer-supported cooperative work*, pages 33-42, Toronto, Canada, November 1992.

[**Inp99**] Inprise Corporation. Entera Enterprise suite. http://www.inprise.com/entera/, 1999.

[**ISO97**] ISO/IEC. *The Virtual Reality Modeling Language (VRML97)*, International standard ISO/IEC 14772-1:1997 edition, 1997.

[**ITI98**] ITI (NCITS). Information Systems - Database Language - SQL (SQL-92). Technical Report ANSI X3.135-1992 (R1998), ANSI, 1998.

[**IYS86**] R.K. Iyer, L.T. Young, and V. Sridhar. Recognition of error symptoms in large systems. In *Proceedings of the 1986 IEEE-ACM Fall Joint Computer Conference*, 1986.

[**JDB89**] V. Jagannathan, R. Dodhiawala, and L.S. Baum, editors. *Blackboard Architectures and Applications*, volume 3 of *Perspectives in Artificial Intelligence.* Academic Press, New York, 1989.

[**JMS95**] H.V. Jagadish, I.S. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *Proceedings of the 14th conference on Principle of Database Systems (PODS'95)*, pages 113-124, San Jose, CA USA, May 1995.

[**JS94**] H. Jagadish and O. Shmueli. *Distributed Object Management*, Chapter: Composite Events in a Distributed Object-Oriented Database. Morgan-Kaufmann, 1994.

[**JZ88**] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, 1988.

[**KFRC93**] R. Kraut, R. Fish, B. Root, and B. Chalfonte. *Groupware and Computer Supported Co-operative Work*, Chapter: Informal communication in organisations. Morgan-Kaufmann, 1993.

[**KK97**] A. Koschel and R. Kramer. Configurable active functionality for CORBA. In *Proceedings of 11th ECOOP'97 Workshop on CORBA Implementation, Use and Evaluation*, June 1997.

[**Kop92**] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 460-467, Yokohama, Japan, June 1992.

[**KR95**] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845-857, October 1995.

[**KSP+95**] T.L. Kunii, Y. Shinagawa, R.M. Paul, M.F. Khan, and A.A. Khokar. Issues in storage and retrieval of multimedia data. *Multimedia Systems*, (3):298-304, 1995.

[**Lad86a**] P. Ladkin. Primitive units for time specification. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 354-359, Philadelphia, USA, August 1986.

[**Lad86b**] P. Ladkin. Time representation: A taxonomy of interval relations. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 360-366, Philadelphia, USA, August 1986.

[**Lam78**] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.

[**LBC+94**] Mik Lamming, Peter Brown, Kathleen Carter, Margery Eldridge, Mike Flynn, Gifford Louie, Peter Robinson, and Abigail Sellen. The design of a human memory prosthesis. *The Computer Journal*, 37(3):153-163, 1994.

[**Lig91**] G. Ligozat. On generalized interval calculi. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 234-240, July 1991.

[**LIT91**] I. Lee, R.K. Iyer, and D. Tang. Error/failure analysis using event logs from fault tolerant systems. In *21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 10-17. IEEE Computer Society Press, Montreal, Que., Canada, 1991.

[**LN93**] Michael G. Lamming and William M. Newman. Activity-based information retrieval: Technology in support of personal memory. Technical report, Xerox Research Centre Europe, 61, Regent Street, Cambridge, UK, 1993.

[**Lop99**] Diego Lopez De-Ipina. Image recognition techniques for identification of users and objects. Personal communication, 1999.

[**Lov91**] L. Lovstrand. Being selectively aware with the Khronika system. In *Proceedings of the 2nd European Conference on Computer Supported Cooperative Work (ECSCW'91)*, pages 265-277, Amsterdam, Netherlands, 1991.

[**LR85**] T.J. LeBlanc and A.D. Robbins. Event-driven monitoring of distributed systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 515-522, Denver, CO USA, May 1985.

[**LS90**] T.Y. Lin and D.P. Siewiorek. Error log analysis: Statistical and heuristic trend analysis. *IEEE Transactions on Reliability*, 1990.

[**LS97**] E. Lupu and M. Sloman. A policy-based role object model. In *Proceedings of the 1st IEEE Enterprise Distributed Object Computing Workshop (EDOC'97)*, 1997.

[**LSBH00**] Sheng Feng Li, Mark Spiteri, John Bates, and Andy Hopper. Capturing and indexing computer-based activities with Virtual Network Computing. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2000)*, Como, Italy, March 2000.

[**LSH99**] Sheng Feng Li, Quentin Stafford-Fraser, and Andy Hopper. Applications of stateless client systems in collaborative enterprises. In *Proceedings of the Fourth International Conference on Enterprise Information Systems (ICEIS'99)*, Setubal, Portugal, March 1999.

[**MA94**] C. Montangero and V. Ambriola. *OIKOS: Constructing Process-Centred SDEs*. Advanced Software Development Series. John Wiley and Sons, 1994.

[**Ma97**] Chaoying Ma. An alarm correlator based on the Cambridge event technology. Computer Laboratory, University of Cambridge, 1997.

[**Mat88**] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of Parallel and Distributed Algorithms*, pages 215-226, 1988.

[**MBS+99**] Ian Marshall, John Bates, Mark D. Spiteri, Chrys Mallia, and L. Velasco. Active management of multi-service networks. In *Colloquium on Control of Next Generation Networks, IEE Electronics and Communications*, London, UK, October 1999.

[**MCC$^+$97**] I. Marshall, J. Cowan, J. Crowcroft, M. Fry, A. Ghosh, D. Hutchinson, M. Sloman, and D. Waddington. Alpine-application level programmable inter-network environment. *BT Technology Journal*, April 1997.

[**MDL93**] S. Menon, P. Dasgupta, and R.J. LeBlanc. Asynchronous event handling in distributed object-based systems. In *Proceedings of the 13th Conference on Distributed Computing Systems*, pages 383-390, Pittsburgh, USA, May 1993.

[**MH93**] Scott L. Minneman and Steve R. Harrison. Where Were We: Making and using near-synchronous, pre-narrative video. In *Proceedings of the 1st ACM International Conference on Multimedia (Multimedia 93)*, pages 207-214, Anaheim, CA USA, June 1993.

[**MHJ$^+$95**] Scott Minneman, Steve Harrison, Bill Janssen, Gordon Kurtenback, Thomas Moran, Bill Smith, and Bill van Melle. A confederation of tools for capturing and accessing collaborative activity. In *Proceedings of the 3rd ACM International Conference on Multimedia (Multimedia '95)*, pages 523-534, San Francisco, CA USA, November 1995.

[**Mic99a**] Microsoft Corporation. Message Queue Server Reviewer's Guide. http://www.microsoft.com/ntserver/appservice/techdetails/overview/msmqrevguide.asp, April 1999.

[**Mic99b**] Microsoft Corporation. Microsoft Office. http://www.microsoft.com/office/, 1999.

[**Mic99c**] Microsoft Corporation. Windows 2000. http://www.microsoft.com/windows2000/, 1999.

[**Mic99d**] Microsoft Corporation. Microsoft ODBC. http://www.microsoft.com/data/odbc/, March 1999.

[**Mil91**] D.L. Mills. Internet time synchronisation: The network time protocol. *IEEE Transactions on Communications*, pages 1482-1492, October 1991.

[**MKL95**] R.A. Morris, L. Khatib, and G. Ligozat. Generating scenarios from specifications of repeating events. In *Proceedings of International Workshop on Temporal Representation and Reasoning (TIME 95)*, pages 41-48, Melbourne, April 1995.

[**MR90**] N.H. Minski and D. Rozenshtein. Configuration management by consensus: An application of law-governed systems. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 44-55, 1990.

[**MS97**] Masoud Mansouri-Samani and Morris Sloman. GEM: A generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), June 1997.

[**MW88**] S. Matsuoka and K. Wakita. Using tuple space communication in distributed object-oriented languages. *SIGPLAN Notices*, 23(11):276-284, 1988.

[**Nel98**] Giles J. Nelson. *Context-Aware and Location Systems*. PhD thesis, University of Cambridge, January 1998.

[**NX93**] R.H.B. Netzer and J. Xu. Adaptive message logging for incremental replay of message-passing programs. In *Proceedings of Supercomputing 1993*, pages 840-849, 1993.

**[OG97]** Open Group, editor. *DCE 1.2.2 Introduction to OSF DCE.* Number Open Group Product Documentation F201 11/97 in DCE Documentation. Open Group, 1997.

**[OMG97]** Object Management Group - OMG. Event Service Specification. Technical Report formal/97-12-11, 1997. ftp://www.omg.org/pub/docs/formal/97-12-11.pdf.

**[OMG98a]** Object Management Group - OMG. Notification Service Specification. Technical Report telecom/98-06-15, 1998. ftp://www.omg.org/pub/docs/telecom/98-06-15.pdf.

**[OMG98b]** Object Management Group - OMG. CORBA Messaging Specification - joint revised submission. Technical report, 1998.

**[OMG99]** Object Management Group - OMG. *The CORBA/IIOP 2.3.1 Specification*, formal/99-10-07 edition, 1999. http://www.omg.com/corba/corbaiiop.html.

**[OPSS93]** B. Oki, M. Pfuegl, A. Siegel, and D. Skeen. The information bus: An architecture for extensible distributed systems. In *Proceedings of ACM SIGOPS 93, 1993*, 1993.

**[Ora99]** Oracle Corporation. Oracle8i Advanced Queuing. http://www.oracle.com/database/features/advque.html, 1999.

**[PN93]** Cherri M. Pancake and Robert H. Netzer. A bibliography of parallel debuggers. *ACM SIGPLAN Notices*, (28):169-186, December 1993.

**[Pur94]** J.M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Language Systems*, 16(1):151-174, January 1994.

**[RDR98]** Devina Ramduny, Alan Dix, and Tom Rodden. Getting to know the design space for notification servers. In *Proceedings of the ACM 1998 conference on Computer Supported Cooperative Work (CSCW '98)*, page 227, Seattle, WA USA, November 1998.

**[Rei90]** S.P. Reiss. Connecting tools using message passing in the FIELD environment. *IEEE Software*, 4(7):57-67, July 1990.

**[Reu80]** A. Reuveni. *The Event Based Language and its Multiple Processor Implementations.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1980.

**[RN96]** Daniel Robey and Michael Newman. Sequential patterns in information systems development: An application of a social process model. *ACM Transactions on Information Systems*, 14(1):30-63, January 1996.

**[RSWH98]** T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1), January/February 1998.

**[SCT95]** Gradimir Starovic, Vinny Cahill, and Brendan Tangney. An event based object model for distributed programming. In *OOIS (Object-Oriented Information Systems) '95*, pages 72-86, London, UK, December 1995. Springer-Verlag.

**[Sha89]** Y.P. Shan. An event driven model-view-controller framework for Smalltalk. In *Proceedings of the Object Oriented Programming Systems and Applications (OOPSLA'89)*, pages 347-352, 1989.

[**SKB+98**] Steve Shafer, John Krumm, Barry Brumitt, Brian Meyers, Mary Czerwinski, and Daniel Robbins. The new EasyLiving Project at Microsoft Research. In *Proceedings of the Joint DARPA/NIST Smart Spaces Workshop*, pages 30-31, Gaithersburg, MA USA, July 1998.

[**Slo94**] Maurice Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4), 1994.

[**SLR94**] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *Proceedings of ACM SIGMOD 1994 International Conference on Management of Data*, 1994.

[**Smi84**] E.T. Smith. Debugging tools for message-based, communicating processes. In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 303-310, San Francisco, CA USA, May 1984.

[**SN92**] K.J Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229-268, July 1992.

[**Sno93**] Richard Snodgrass, editor. *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, USA, June 1993.

[**Sno95**] Richard Snodgrass. *The TSQL2 Query Language*. Kluwer Academic, 1995.

[**Sun97a**] Sun Microsystems. Java Remote Method Invocation Specification. http://java.sun.com/products/jdk/rmi/, 1997.

[**Sun97b**] Sun Microsystems. The Java Abstract Window Toolkit. http://java.sun.com/products/jdk/awt/, 1997.

[**Sun98**] Sun Microsystems. *ONC+ Developer's Guide*, 805-4034-10 edition, 1998.

[**Sun99a**] Sun Microsystems. Java Foundation Classes/Swing. http://java.sun.com/products/jfc/, 1999.

[**Sun99b**] Sun Microsystems. JDBC Data Access API. http://java.sun.com/products/jdbc/, 1999.

[**Sun99c**] Sun Microsystems. The Java 2.0 Language. http://java.sun.com/products/jdk/1.2/java2.html, 1999.

[**SY85**] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3:204-226, 1985.

[**TA87**] Kuo-Chung Tai and Sanjiv Ahuja. Reproducible testing of communication software. In *Proceedings of IEEE COMPSAC'87*, pages 331-337, 1987.

[**Tal98**] Talarian Corporation. SmartSockets. http://www.talarian.com/products/smartsockets/smartsockets.shtml, 1998.

[**Tan91**] J. C. Tang. Findings from observational studies of collaborative work. *International Journal of Man-Machine Studies*, 34:143-160, 1991.

[**TCG+93**] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors. *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings, 1993.

[**Ter97**] Paolo Terenziani. Integrating calendar dates and qualitative temporal constraints in the treatment of periodic events. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):763-783, September/October 1997.

**[TGD97]** D. Tombros, A. Geppert, and K. Dittrich. Semantics of reactive components in event-driven workflow execution. In *Proceedings of the 9th International Conference on Advanced Information Systems Engineering*, June 1997.

**[Tib99]** Tibco Software Inc. TIB/ActiveEnterprise. http://www.tibco.com/products/active_enterprise/index.html, 1999.

**[TIN96]** Telecommunications Information Networking Architecture Consortium - TINA. TINA Notification Service Description, July 1996.

**[Tsa83]** Michael M. Tsao. *Trend Analysis and Fault Prediction*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1983.

**[WC96]** J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

**[WF92]** Y.M. Wang and W.K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 147-154, October 1992.

**[WFD94]** S. Whittaker, D. Frohlich, and O. Daly-Jones. Informal workplace communication: What is it like and how might we support it? In *Proceedings of Conference on Human Factors in Computing Systems (CHI'94)*, pages 130-137, 1994.

**[Whi96]** S. Whittaker. Talking to strangers: An empirical evaluation of factors underlying electronic collaboration. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96)*, pages 409-418, 1996.

**[Wit88]** Larry D. Wittie. Debugging distributed c programs by real time replay. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 55-67, Madison, WI USA, May 1988.

**[WJH97]** Andy Ward, Alan Jones, and Andy Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42-47, October 1997.

**[WMLF98]** P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T-Spaces. *IBM Systems Journal*, pages 454-474, 1998.

**[WS96]** S. Whittaker and C. Sidner. Email overload: Exploring personal information management of email. In *Proceedings of ACM CHI'96 Conference on Human Factors in Computing Systems*, pages 276-283, 1996.

**[WSKS97]** S. Whittaker, J. Swanson, J. Kucan, and C. Sidner. TeleNotes managing lightweight interactions in the desktop. *ACM Transactions on Computer-Human Interaction*, 4(2):137-168, 1997.