Metadata support for connecting application components asynchronously

Oliver Seidel

PhD Dissertation

1999

Churchill College, University of Cambridge

— Mit Dank an meine Eltern —

PREFACE

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Ken Moody for his example, advice and encouragement during the last four years.

I would also like to thank the members of the OPERA group for many inspiring discussions, and for providing such a friendly working environment. My particular thanks go to Chaoying Ma and Richard Hayton, who were the most experienced researchers that I shared an office with. Alan Abrahams and Alexis Hombrecher have taken a lot of time to carefully read the drafts for this dissertation and have provided valuable feedback – thank you.

Many thanks go to my parents, without whose support and encouragement I would have never come here to work towards a degree. I would also like to thank my girlfriend Barbara Lieske for enduring my absence.

The research carried out for this thesis was funded by the University of Cambridge Computer Laboratory Supporters' Club.

ABSTRACT

The technological advances of recent years have led to the proliferation of networking and inter-networking: while networking originally was reserved to large installations that could afford the associated hardware and service costs, it now has wide application in firms and even some homes. In the commercial context, one can often find installations that are pervasively networked, but separated from the outside world by a security gateway. In the consumer context, it is almost exclusively the case that the technology provides asymmetric channels into the home. While the current crop of applications uses a client-server model with synchronous invocation, the next generation will involve a larger number of simpler modules with more specific purposes. Wireless reporting and home monitoring as well as business process notification are likely to use asynchronous technology. These systems will only find acceptance if they are sufficiently automated and robust. They also need to be flexible enough to cope with a merging of previously independent management domains.

In this dissertation I argue that the metadata describing those systems must be readily accessible in machine-readable form from within the system. If information describing typing, the application and the configuration is available, greater flexibility can be achieved, automation becomes possible and better monitoring improves robustness. I argue for a way of structuring data and metadata that allows the most important parts to be treated with the reliability and the associated performance trade-off of transactions. This structuring also makes the boundaries of autonomous components explicit. The resultant architecture is suitable for a range from simple to complex systems by allowing the omission or inclusion of appropriate functionality.

During this research an appropriate format has been chosen to represent the metadata. The specification is available in form of an object model, with a concrete grammar, and a framework to produce the executable code from the documentation. Several serialisation regimes are available, trading the speed of bulk operations against that of partial updates and trading speed and size against hardware independence. The serialisation regime can be changed on a per-invocation basis. Significant effort went into the persistence component, which features clean interfaces to flat or hierarchical variable-size segments and integrates with a high-performance logging service. All components have been designed to provide transactional guarantees, but the actual transaction manager implementation only supports a limited number of execution paths.

The first phase of the project established a flexible object persistence mechanism, with special attention to the needs of event stream storage. The metadata of events was then specified by establishing an object database schema. Experiments to define, store and evolve an event type hierarchy have been carried out successfully in this framework. In addition a search directory system has been tested which could provide yellow pages support for application-level event services. This thesis proposes a metadata based architecture for connecting application components asynchronously using events as glue, and suggests where component boundaries can be drawn most effectively.

TYPOGRAPHIC CONVENTIONS

sample text	description of usage
Example	foreign expression
`Example`	computer source code
Example	computer science terminology
‘Example’	unusual usage of a word
‘Example phrase’	unusual usage of a phrase
Example	name of a product
Example	name of a company
Example	name of an international standard
Example	name of an international standards organisation

CONTENTS

1. INTRODUCTION

This dissertation presents methods for improving the robustness and flexibility of asynchronous middleware through the aggregation and provision of metadata. Standardised access to a machine-readable representation of the system configuration allows more informative management tools to be built, allows automatic re-configuration to be put in place, and as a result, simplifies the implementation of fail-over mechanisms. The subsequent chapters will investigate the following topics:

Middleware

The middleware review in chapter 2 gives an overview of the many things that middleware means to many people. It elaborates on the state of the art for remote invocation, comparing the reliability, performance, flexibility and real-world use of different systems.

A look at asynchronous messaging reveals that different strands of this technology have been around for a long time, but are converging to incorporate each other's abilities. Metadata storage and resource discovery are reviewed in a section on traders for system configuration information and directory technology for storage of user data. At the end of this review is an investigation into transaction processing for the provision of guarantees and database connectivity for simplified engineering of component architectures.

Each of these technologies is a success of their own and understanding their strengths and shortcomings provides a good perspective for the architecture presented in the following chapters. The arguments used to advance particular techniques and the motivation behind their development form themselves a good body of support for the concept presented here.

Object Databases

Where the packaging of object state is an important theme in middleware, object databases face the same problems, but in the context of a different set of assumptions. Chapter 3 portraits the influences that led to the development of Object Oriented Database Management Systems and provides an analysis of different approaches to persistence, evolution, and access taken by these systems.

The development of these systems is followed by an investigation into the ODMG database standard. It is shown how its object model accommodates use with several programming languages. Finally its query language is explained, which tries to preserve the best parts of the successful *SQL* standard for relational database management systems. Databases typically maintain and manage large amounts of data, which leads to the increased need for optimisation. Databases attempt to amass as much metadata on their contents, as they possibly can: typing, structural and statistical information are collected and exploited in query optimisation, index creation and data placement.

Since database management systems have started incorporating this metadata-based technology 30 years ago and new methods have constantly been discovered, database technology is relevant to this thesis. The event architecture portrayed in this dissertation is actually based on some parts of the ODMG object database technology and this chapter explains what the database advantages are and how they can be used to leverage the derived event architecture.

Event Directories

Against the background of event technology and the capabilities of the ODMG standard, chapter 4 on Event Directories spells out the need of infrastructure for event services. It points out the similar justification for an ODMG database schema and an Event Type Directory. It establishes the requirements of storing typing information and focuses on storage and subsequent provision of configuration information. It shows how broker functionality can be implemented with the use of this information and states the metadata needs of gateways for the interworking of domains. With up-to-date brokers established, the description then finishes with the distinction between white-pages services (brokers) and yellow-pages services (search engines).

Since the aim of the architecture presented in this dissertation is to simplify the use of supporting services so that programmers can concentrate more of their efforts onto developing applications, the chapter on event directories highlights similarities between the approaches to metadata storage in other usage domains. The methods presented are ordered by the volatility of the data that they store, because that affects their design. The motivating examples, which are well established in the literature, are discussed in support and in explanation of the methods chosen in the context of the CEA.

Wide-Area Event Services

The previous chapter has described how various parts of the event architecture can be documented in a machine-readable form. Chapter 5 goes on to investigate Wide Area Event Services. It describes how new problems arise, simply through a wider distribution. With an eye on resource discovery, interworking and access control, this chapter distinguishes between the necessary white-pages service and the convenience of a yellow-pages infrastructure.

The chapter addresses the problems that arise when domains have autonomous administration, but wish to inter-operate. To make inter-operation feasible, it describes how resources can be discovered across domain boundaries and how a domain can control the amount of internal information exposed to the outside. This leads to a detailed discussion of the capabilities that a trader must have, in order to facilitate inter-operation in the face of schema evolution and component failures.

The chapter then investigates the mismatch between differing systems, and makes an attempt to overcome some of the problems that arise when the interworking of heterogeneous event systems is required. The chapter concludes with a description of gateways, which allow for a more controlled information flow between domains.

These inter-operation services presented in chapter 5 only become possible once sufficient metadata is available. Automating these services allows for fall-backs, reconfiguration and buffering. The increased robustness of this architecture over synchronous special-purpose applications is the driving motive behind collecting the metadata in the first place.

Distribution and Federation

The presentation of schema evolution problems in a controlled domain leads to chapter 6 on Distribution and Federation. The chapter first discusses the development approaches for federated systems, and how participants in a federation can balance their autonomy against the desire to inter-work. Then it highlights the particular case of the Cambridge Event Architecture.

A model for relating the different schemas in a federation is proposed and schema evolution is discussed. For schema evolution, the chapter presents a list of applicable changes that can occur to the schema and then addresses in detail issues of naming, changes to the class hierarchy and resulting name clashes. The solution to some of these problems is the recording of higher level information, namely, which operations have been applied to effect the changes to the schema. The next topic addressed in the context of federation is access control. The chosen scheme is explained and since it is a federated system itself, it is a good match for the proposed architecture. Then the chapter addresses the problem of different service levels and how they can be labelled. Different domains may well provide differing guarantees and translating the information describing those guarantees between domains is a non-trivial task. This chapter argues for making these labels part of the architecture, so that divergence of labels and meanings can be averted.

The chapter concludes with an analysis of performance requirements and a resulting recommendation for partitioning the metadata according to use. The aim of this chapter is to highlight the differences between performance and management issues in different scopes. It shows how distribution beyond a single management domain introduces greater complexity, which quickly becomes unmanageable without automation. It then makes a case for increasing the availability of metadata to aid this automation.

Experimental Framework

The CEA is a multi-faceted research collaboration, ranging from transmission mechanisms to human computer interfaces. In order to test and measure the concepts and ideas from this dissertation, a number of experiments have been conducted.

Chapter 7 gives an overview, and elaborates on the implementation of the aspects investigated previously. The chapter points out a number of applications that benefit from event technology and then highlights the difficulties that current implementations of these applications face. It also lists the benefits to be expected from increased metadata availability.

The following sections describe the implementation of these event services, with the transmission mechanism and the interfaces of the event sources and sinks. A section on arbitration and translation describes how gateways can easily be constructed from the available components and a section on directory services describes the repositories for increasingly volatile data, namely schema repositories, traders and search engines. A section on persistence describes the current storage implementation and the reasoning behind its design. The section on query processing describes the mechanism for instantiating arbitrary objects with user-supplied methods inside the object database. A further part of that database is described in the section on the distributed transaction manager. The final section describes the access control mechanism for events, storage and metadata access.

These descriptions all include comment on the difficulties encountered during the construction of this system and provide valuable insight into design trade-offs.

Case Studies

While select experiments were used to confirm the appropriateness of individual aspects of this architecture, chapter 8 presents a number of Case Studies. The aim of these scenarios is to illustrate how the components provided by the architecture can be combined to serve as a flexible environment for the development of reliable distributed applications.

The first study discusses an example of monitoring communications networks, which has been investigated extensively in collaboration with a telecommunications firm. Special notice is taken of the way in which event technology alleviates boundary problems between different firms. The second study on workflow coordination highlights the advantages of asynchronous operation for this data-driven application. The third study is based on previous experience with a multi-media presentation system. Of particular interest in that example is the ability to combine differing information sources, to use events as a means of synchronisation, and to show how the previous implementation would benefit from the increased availability of source statistics and usage information. The last case study concerns the case of inter-operation between digital libraries.

With the access control mechanism tightly bound to event technology, this example concentrates on domain management and detached working.

Together, these studies confirm that increased knowledge about the configuration and state of the system simplifies automation and increases end-user transparency. They show how the components described in the previous chapter inter-work to provide a framework for reliable messaging in the face of evolution.

Conclusions and Further Work

The final chapter 9 summarises how the provision of metadata on the abilities and configuration of the asynchronous messaging infrastructure leads to simplified inter-operation, makes it possible to allow for dynamic evolution and increases robustness by allowing better judgment of the quality of data. The research has shown how the functionality ought to be segmented into components and in this chapter some recommendations are given for further work based on this structuring.

One suggestion provided for further work is the locally developed prototype for the active house. In this context, experiments to test evolution and access control might involve a number of houses and maybe some simulated public utilities. Different strategies for finding a consensus schema in a federation could be tested and event retrieval and replay could play a growing role.

2. MIDDLEWARE AND MOBILE APPLICATIONS

In order to cope with the increasing diversity of applications and computing platforms, a new layer of abstraction has been introduced to facilitate maintenance and re-usability. This chapter gives an introduction to the meaning of the word middleware. It then describes the subset of middleware functionality that this thesis addresses and investigates examples from the literature.

As the name suggests, middleware is used as an adaptive layer between applications and other applications or resources. Middleware defines styles of interaction and typically provides support to conform to the method.

Middleware support for a particular style of interaction usually comes in form of a description language and some accompanying software services. Features often found in middleware are:

- persistence services to store complex state
- transaction functionality
- object oriented distribution for remote method invocation (synchronous messaging)
- event services (asynchronous messaging)
- directory services
- mobility / agent technology
- quality of service / real time control interfaces to transport layer
- reflection for run-time adaptation, typically in cooperation with the Quality of Service (QoS) interface
- multimedia platform[1]
- groupware platform[2]

[1] Such a platform could provide pre-packaged objects for the creation of video windows, search dialogs, search result presentation, etc.. An example can be found in [Bat94].

[2] A groupware platform would provide operations for the management of groups: it would provide operations to join / leave a group, permit / revoke / delegate access rights and probably integrate with authentication services which are going to be used for signing and authorisation.

2.1 Overview

2.1.1 Research Emphasis

The research concerns the use of OODB technology to characterise event type systems. It describes extensions to the locally developed distributed event system with directory, versioning, transaction and persistence services. This chapter will describe some implementations of the chosen functionality and how they differ from the presented approach. Emphasis will be placed on:

Messaging Middleware. typically comes in two flavours: low latency and high reliability; the local system has focussed on low latency messaging, but the research investigates a more reliable method.

Directory Services. for both end user information (user, resource and authentication information) and system level information (name/OID mapping, routing information, object types, semantic descriptions, versioning). The research provides a general purpose database that is used as a reliable directory for both kinds of data. Appropriate interfaces are discussed.

Transaction Processing. originally in databases, but now as part of reliable message oriented middleware or quite separate for reliable computation. The research investigates the suitability of modern transaction models for use in messaging middleware.

Database Connectivity. e.g. providing persistence to applications, or providing front ends (e.g. a World Wide Web (WWW) interface) to (legacy) database applications. Based on the ideas of the Multi Service Storage Architecture (MSSA) [Lo94, Tho90], a reliable object store is constructed and accessed through a modern interface [Cat97].

The following sections will give a short introduction to each topic and then describe research prototypes or commercial products implementing the technology and their lessons learned.

2.1.2 Distributed Object Systems / Remote Execution

These systems have in common that they shield the user from the details of how to talk to different networking protocols and hardware, how to represent data on different hardware platforms, and how to present it in different programming languages.

Objects

The object paradigm consists of the idea that an object is the representation, rather than the description of an entity. Object oriented programming languages support this model and typically feature the following 3 properties for each object instance:

Behaviour. The behaviour of objects is defined by the methods that the object makes available to the outside.

State. The state of the object consists of the contents of its attributes. These values may be observable only implicitly through access methods. Computed or otherwise derived attributes are not part of the state of the object.

Identity. Two objects have the same identity if modifications to either one of the objects effect exactly the same observable changes on both. This differs from equality, where two objects are indistinguishable by mere observation, but modifications on one are not observable in both. Object systems often provide means for using an Object Identifier (OID). This OID may be a pure name, or a memory address or a persistent storage identifier.

These properties are typically described in some description language, which will define the *type* of the object. It is common to find the object types available to a program arranged in an acyclic directed graph called the *class hierarchy.* Arcs are called *inheritance relations* and are used to extend simpler objects (super types) with specific functionality to yield a more complex type (subtype). Frequently there is the additional requirement that the graph must be a tree (that there exists no confluence inside this graph). An inheritance graph with joined arcs is called *multiple inheritance* graph, while a tree structured graph is called *single inheritance* graph.

One property often found in object systems is *polymorphism* (see glossary). If a parent object type in a class hierarchy has a certain ability (e.g. to print its contents), then this ability is inherited by all derived types. An object instance of a derived type has a separate type from its parent and a function accepting its parent type will also accept the object itself. The function is thus called *polymorphic.*

Some systems separate the hierarchy of object types and the of *interfaces.* Just like a class declaration, an interface also defines attributes and method signatures. But while the hierarchy of classes is a guide to the compiler on how to construct executable code, an interface is not bound by this restriction. It is common that the hierarchy of interfaces allows multiple inheritance while a hierarchy of classes in the same language system forbids multiple inheritance.

2.1.3 Standards for Distributed Computing

The international standard for the description of distributed object systems is the ISO/ITU-T *Reference Model for Open Distributed Processing* (*RM-ODP*), [ITU95]. It defines a common terminology and raises awareness for all the critical issues in distributed systems design. All current middleware can ultimately be described in the terms of *RM-ODP*.

Besides the all-encompassing *RM-ODP*, at least three industrial consortia provide industrial standards on distributed computing. Each has a different focus, but the issues that they address are the same.

OSF

The Open Software Foundation (OSF) is the industrial standards organisation of the traditional distributed systems vendors. The OSF has now changed names and is called the Open Group. However, the acronym OSF remains. The mark of the Open Group is the 'X' and protocols or conventions published by the Open Group are often prefixed with 'X/Open'.

The Open Group provides middleware in the classic sense, but has not yet addressed object oriented facilities. It currently bases its interfaces on procedural access paradigms, but commonly allows for specification of a particular source. This is the basis for object oriented use of their framework.

OMG

The Object Management Group (OMG) is an industrial standards organisation comprising more than 700 companies. The goals of the OMG are very ambitious in that it aims to provide a complete set of specifications on how to deal with distribution itself, but also all issues arising in the context of distribution. Namely, it standardises services for Association, Events, Garbage Collection (they call it life cycle services), Naming, Notification, Persistence, Security, Trading, Transactions, and a lot of other related areas.

TINA

The TINA Consortium, with their *Telecommunications Information Networking Architecture* (*TINA*), as specified in [Con94], focus mostly on QoS issues.

2.1.4 Implementations

While the standards described above provide good guidance, their shortcomings are not immediately obvious. This subsection will look at implementations of distributed systems infrastructures and highlight their particular strengths and weaknesses.

OSF *DCE*

The Open Software Foundation *Distributed Computing Environment (DCE)* [Mau99] is a mature set of specifications for building robust and large distributed systems. *DCE* provides the Remote Procedure Call (RPC) service, the *Cell Directory Service (CDS)* and *Global Directory Service (GDS)*, the *Security Service*, *DCE Threads*, *Distributed Time Service* (DTS), and *Distributed File Service (DFS).* A subset of these services, namely the *DCE RPC*, *CDS*, *DCE Security*, *DTS* and *DCE Threads*, are called the 'core services' and are required for a system to be called 'DCE compliant'.

Mature implementations of the 'core services' have long been available from large systems vendors such as International Business Machines (IBM) and Digital Equipment Corporation (DEC). *DCE* has traditionally been available on mainframe and mini computers. The usage was typically to design applications *in toto*, rather than putting emphasis on re-use of components. The *DCE* services are viewed as part of the infrastructure, rather than being situated on top of it.

A lot of work has been done to provide the functional structure of the *DCE* components and formal justification of the protocols involved. Although the *DCE* is not explicitly object oriented middleware, its structure does not prevent an object oriented structuring of a *DCE* system.

CORBA

The *Common Object Request Broker Architecture* (*CORBA*) defines a standard[Obj98a] for providing a distributed computing environment. At its core, it provides the *CORBA* Interface Definition Language (IDL), language bindings on how interfaces defined in IDL map into different programming languages and a run-time environment. Further extensions to *CORBA* follow in the form of *services*, the interfaces to which are specified in IDL and the semantics of which are described in the accompanying *CORBA* documents.

CORBA features objects, which possess at least *state* and *behaviour*. The state of a *CORBA* object is the contents of its variables. Its behaviour is the semantics behind each of its published method signatures. The *identity* of *CORBA* objects can be established with a call to the interface `CosIdentity::IdentifiableObject`. Maintaining the illusion of identity while using replicas of the same object in separate locations for quick read access, especially in the case of detached operation, is a hard problem. This problem affects event technology to a lesser degree, since events are operated on with restricted operations: the core data describing the occurrence creating the event is immutable. The event object will only be changed by attaching and changing metadata, possibly including a transmission history. The *CORBA* standard itself is not very explicit about the permission or requirement to provide object identity. However, one of the *CORBA* services (see section 2.2.3) mentions in passing that *CORBA* objects do not have identity.

The *CORBA* class hierarchy is a single inheritance hierarchy and is used for defining types, as well as defining arguments.

Given a *CORBA* object reference, a program can perform method invocations on that object. The run-time system then either resolves the reference to an object within the same *process*[3] or determines the location of the referenced object, marshals the arguments, ships them to the location of the object and performs the invocation, returning the results to the caller afterwards.

Microsoft *DCOM*

Microsoft Distributed Component Object Model (DCOM) [BK, Red97] is the distributed version of Microsoft Component Object Model (COM). COM defines a convention for *binary interfaces* [Rog97], which are closely modeled after the C++ virtual function table layout. Each interface is a collection of functionally related methods on a particular class. These objects can be accessed by any language that is designed to adhere to this memory layout or is provided with a suitable wrapper library.

Users never obtain a handle on the object, they obtain handles on one of the interfaces of the object. Each COM object must provide the standard interface to query its services (`IUnknown`, which contains the methods `QueryInterface`, `AddRef`, `Release`).

One can interpret DCOM as session oriented, since each object keeps a reference count and keeps timeouts for when the client dies. *CORBA*, on the other hand extracts that behaviour into the life cycle services, which are an add-on to the core functionality. While there is a lot of scope for optimisation within a session oriented system, the documentation for DCOM only describes simple ideas, like caching.

Feature Comparison

Diagram 2.1 compares the RPC functionality of the three systems described above.

[3] In this context, *process* means protection domain and in local memory.

Feature Comparison

DCE	CORBA	DCOM
procedural	o oriented	binary iface
no classes	iface hierarchy	indiv. binary ifs
multi language	multi language	multi language
multi system	multi system	single system
pass-by-val	by-val / by-ref	by-val / by-ref

Fig. 2.1: Feature Comparison of *DCE*, *CORBA* and *DCOM*

Java RMI

The Sun Microsystems *Java* Remote Method Invocation (RMI) mechanism differs from *CORBA* in a several respects:

- RMI is a single language mechanism.
- RMI supports the same inheritance mechanism as *Java*, namely single inheritance.
- *Java* objects have identity within one *Java* Virtual Machine (JVM), RMI achieves identity by requiring unique identification of the machine and the object within that machine.
- RMI moves state across process boundaries via the *Java* Object Serialisation specification, which allows for class evolution by tagging each field in the output stream with its name, imposing an ordering on the elements in the output stream and allowing for omission of input stream items at the end that reconstructs the object or parameter.
- RMI behaviour is declared in *Java* IDL and defined in *Java* programs.

2.2 Events

Events have a long history in internal systems tasks such as windowing or database triggers. In those environments, a high volume, no failsafe approach is implemented, since the coupling between the components is very tight and delivery order is implicitly equal to notification order. Another established use of events is in message queuing, where reliable delivery is provided through transactional techniques. In short, the capabilities and limitations of event technology as opposed to RPC can be summarised as follows:

Asynchronous Messaging. The important distinction between Events and RPC is that event communications are asynchronous. Events get sent and the program execution immediately returns to the sending program without the opposite side necessarily having received the message just yet. RPC (or on objects RMI), which is synchronous, delays the execution of the calling program until the result (which may be empty) returns. Events therefore provide high throughput, because further events can be sent during the time that the program is not stopped. Since this technology is frequently used in multi media applications, it is optimised to reduce the latency of delivery. If event communications are used to simulate RPC, then events from the client to the server will be named requests and sent along one channel, while the corresponding answers are going to be sent as events in a separate channel that is directed from the server to the client. An event system does not necessarily guarantee to preserve ordering.

Sender/Receiver Decoupling. Event systems decouple the sender and receiver. In case of event notification, the data is marshaled into a universal format and made available to whoever wants it. The receiver on the other end does not necessarily need to know where the data is coming from either (this is quite similar to RPC), and finally, the event can be delivered to several destinations simultaneously (except in message oriented middleware where 'exactly once' semantics are promised).

Time Stamping. Since events signify an occurrence at a particular point in time (typically the completion time of something), they are marked with that time stamp. This is different from most RPC systems. An RPC system might utilise a timestamp internally to ensure 'exactly once' semantics, but this timestamp would not be made available to the user.

Quality of Service. Again, synchronous invocations are processed on a best effort basis (although this may be viewed as changing with electronic commerce taking off and vendors reserving some of their WWW server capacity for buy transactions). Modern event systems, especially when distributed, typically allow for prioritisation of events.

Uniform Typing. Modern RPC systems declare the interface of a service. Such a declaration includes the signatures of the available methods. Event systems typically go further than compile-time declarations and allow the type of the event or of a particular event source to be inquired at run-time. It is also possible to dynamically access events of a type that was not known at compile time. The types of these events generated by a particular source can then be stored in directory services in the same way in which distributed object systems provide brokerage. While it is possible to simulate these advantages with RPC by sending type `Any` items and to provide an internal encoding, having the infrastructure with type support improves robustness.

Uniform Proxy Service. Since events can be sent once and received by many, scalability in the wide area becomes an issue. As events are constructed in a uniform way, they can be duplicated easily and a single proxy service can handle an unlimited number of event types. These proxy services can be arranged in a hierarchical fashion, with each step trading off a small cost in delivery latency against further scalability. Multicast capabilities of the network may be used, but current implementations are limited by the need to rapidly create new multicast groups. Also, if the event system wishes to provide delivery guarantees, the multicast system must be reliable, too.

Uniform Storage Service. Drawing on the uniform representation of events again, they can be stored in a general purpose store, either with or without database qualities. If the store understands that structure, it can index on particular attributes [SB98].

Uniform Processing Service. Generic services can be constructed to process arbitrary events (something that would have been unthinkable in the RPC world, but has recently become possible in remote method invocations through the use of reflection). Such a service might filter on type or content, it might mine a stored collection of events, or even process continuous streams of events and generate further, 'higher level' events (for examples of such technology, please see [MSS95, MB98]).

This section will describe several implementations, along with their focus.

2.2.1 Message Oriented Middleware (MOM)

Message Oriented Middleware (MOM) facilitates a programming style known as 'messaging and queueing'. The main difference to the event and tuple systems (described in section 2.2.2) lies in its emphasis.

The typical setup of MOM consists of a set of message type definitions and a set of directed pathways. Each pathway has an associated list of allowed message types and an 'undeliverable' container.

Decoupling. MOM systems do not require the sender and receiver to be aware of each other. The sender of the message only submits it into a named queue. This queue may either be connected to the receiving application, or it may be connected to an intermediary processor that either duplicates or filters or processes the message in some way.

Direction. MOM systems are uni directional.

Distribution. MOM systems are distributed by design. Their task is to reliably transport messages across physical distances and/or protection domains.

Quality of Service. MOM systems do not offer guarantees with regard to timing, but they allow priorisation of some messages over others.

Type System. MOM systems may provide a type system, but that is not a prerequisite.

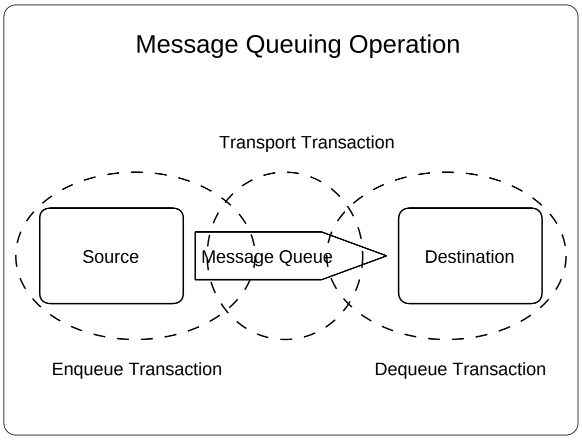

Fig. 2.2: Message Oriented Middleware

Synchronisation. MOM systems are decoupled across both space and time. This means that only the queuing system and the application program need to be instantiated at the time that the message is transmitted. There is no need for the receiving program to be running at this time.

Transactional Guarantees. MOM systems perform a transaction to enter a message into the queue and another transaction to extract the message from the queue (as shown in 2.2). This requires the programs at both ends to be transactional and hence ensures 'exactly once' semantics. If the endpoints of the queue are in separate protection domains, a third transaction must be performed to transfer the message from the source side to the destination side.

Originally, MOM was used to reliably connect application programs to a centralised database server. This style of operation only required pathways of length one, it only used the `put` and `get` operators and the message transmitted was free form text representing a human readable database query. To this date, all major database management systems ship with a queuing system front end to handle this requirement. But queuing systems are now also used as middleware, facilitating reliable communications between non-database distributed applications.

IBM *MQSeries* and IBM *Message Broker*

The IBM *Message Queuing Series* of software products is probably the oldest kind of middleware. First sold in Q4/1993, *MQSeries* offers the facility to set up named queues and transactionally transmit byte arrays. The named queues can be connected in a directed graph and an external transformation procedure can be applied to a message at the delivery node. While other messaging systems set up channels with the ability to transmit messages of one specific type, any *MQSeries* queue can transmit messages of all types.

The value of *MQSeries* lies in its transactional guarantees and its very widespread implementation: *MQSeries* is available on 40+ platforms. Transactions in *MQSeries* can be controlled by the built-in transaction processing monitor, or an external X/Open (DTP) XA[4]-compliant transaction processing monitor.

The communications model of *MQSeries* decouples senders and receivers of messages in space, time and identity. This means that a sender submits its message to a queue, but does not need to know who will receive it or when. It also means that a receiver extracts a message from a queue and does not know where it came from.

The four operations supported at the client are blocking and non-blocking `take first`, as well as blocking and non-blocking `take cursor`. Cursors are defined to range over a `scan` expression, which can select a group of messages either by message identity (for segmented messages), correlation identity (=reply information) or group identifier (=application name). Mixed selection is not possible.

Messages in *MQSeries* feature several fields in their headers. These fields are by design filled in at the sending station. Since a sending station is controlled by an operating system from a long list of possible options, the integrity of the message cannot be assumed stronger than that of the weakest operating system of that list. Although a mechanism is conceivable in which each receiving *MQSeries* server authenticates the operating system (and the owner of the machine) of its sending partner, this functionality is currently not available.

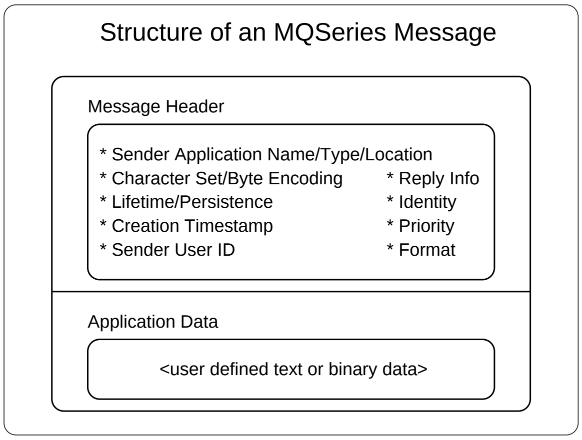

Fig. 2.3: IBM MQSeries Message

But even discounting malicious attacks, most fields of the message header are filled in through human design. This particularly affects the `message format`, `message identity` and `application name` fields.

A second problem is posed by the transactional guarantees, combined with asyn-

[4] 'XA' is a term from the X/Open transaction processing standard and stands for 'Cross Memory Access', as opposed to a remote procedure call.

chronous processing. The queuing system makes a guarantee to the submitting application that the message will be handled. At this point in time, it is not clear that a receiver for the message can actually be found. To cope with this problem, *MQSeries* offers to collect undeliverable messages in a user specified error queue.

A final point of interest is *MQSeries*' support of a priority field. While messages from a single source to a single destination are guaranteed to be delivered in submission order, changing the message priority can break this rule.

While all of the above may make *MQSeries* seem archaic in comparison to modern middleware, it probably is the most widely used MOM product. Its yearly turnover amounts to the order of £100m (in 1999), which is more than 60% (in 1999) of the messaging market.

MQSeries derives its relevance from its dominant position in the financial sector. In this context, it is used to send structured text messages. The messages would contain *Structured Query Language* (*SQL*) statements for processing by a central database or Electronic Data Interchange (EDI) / Extensible Mark-up Language (XML) messages for inter-business communications. *MQSeries* is delivered with two pre-defined message formats (types): a control message type, which gets interpreted by the system itself and a text type, for which a transformation procedure is built into each receiving end. Given the source and destination character encodings, this text is processed in the manner most convenient to the user.

Building on top of this reliable, asynchronous messaging graph, IBM provide *Message Broker*. This product uses the `message format` and `application name` fields to identify messages and provides a uniform manipulation platform. *Message Broker* features a scripting language and the ability to translate messages with user supplied functions.

The *Message Broker* must have access to the structure of a message. Since the message itself only contains the user supplied `message format` label, a separate type repository must be used. This historical separation of the transport and processing layers leads to a loose coupling which complicates generic processing of messages and allows for human errors to introduce type mismatches.

With the *Message Broker* package comes a set of tools that do generate stubs and insert the type into the type repository. The types supported are base types (e.g. `int`, `string`, `money`, ...) and constructor types (e.g. structures and lists).

The event systems described in 2.2.3 have overcome this problem by integrating more metadata into the basic infrastructure.

JPMQ

The *Java* Persistent Message Queue (JPMQ) service [SZ98] is a persistent message queuing service interface for the *Java* programming language. It is a research prototype and hence not bound by commercial necessities. While other tuple systems offer connectivity across ranges of heterogeneous platforms and programming languages, JPMQ is based on the *Java* programming language and is hence limited to only one language and only to those systems that support the run-time environment. Its functionality extends beyond that of conventional message queueing services by providing:

Complex Types. JPMQ supports tuples whose data type is derived from one specific class definition (which is delivered with the package). Since the *Java* type system is very rich and can be interrogated at run-time, JPMQ supports much stricter typing with greater flexibility for run-time modifications than other systems.

Automatic Priority Assignment. Each JPMQ message is a *Java* object. JPMQ allows for functions to be attached to each channel that take messages of a particular type and compute a priority.

Application-specific Dequeue Ordering. The channel specific ordering may not satisfy the needs of all applications sharing a channel. To alleviate this problem, JPMQ makes it possible to define a dequeue ordering function for each dequeue operation.

Selective Dequeue Operations. Since the underlying storage system for the queue manager is a relational database management system (RDBMS), it is very easy to specify a group of messages inside a queue and retrieve those in a single transaction. JPMQ provides this facility and hence is able to perform content dependent dequeuing.

Concurrency Control for Multi-user Environments. Given that JPMQ allows a user to specify a set of messages to dequeue, *isolation* must be ensured amongst several concurrent processes accessing a shared queue. JPMQ allows for the specification of the isolation levels 'none', 'ordered' and 'block-wise'.

While JPMQ contains properties of a queuing system, a tuple space system, an event system and a relational database, its name associates it with queuing systems. However, given its close ties to *Java*, it is questionable whether it can actually be put to use where other queuing systems are used (other tuple systems are used to 'glue' heterogeneous systems together). Since JPMQ has much better typing than IBM *MQSeries*, it could very easily support a powerful message broker and compete with persistent event systems.

2.2.2 Tuple Spaces

Tuple space systems are based on the idea that a central server hosts a bag of tuples, the 'tuple space'. Several simultaneous client processes have available a very small number of primitive operations to interact with the tuple space:

put. atomically insert a new tuple into the tuple space

get. atomically fetch an existing tuple from the tuple space (the tuple space does not contain the item afterwards)

copy. atomically copy an existing tuple from the tuple space (the tuple space does contain the item afterwards)

The use of the English words `get` and `copy` treats the items as if they were physical objects rather than information constructs. This is the intended use, since it conveys the intuitive semantics of the two operations.

While the `put` operation is always synchronous, the `get` or `copy` operations can be either synchronous or asynchronous. Synchronous operation means that they block the execution of their calling program until the desired data is available. The opposite behaviour is asynchronous operation, which returns immediately, possibly returning a failure code indicating the unavailability of data.

Each tuple has a type. The type of the tuple is always that of a structure containing one or more fields. Each of these fields has a type itself, chosen from a list of base types. The fields are not named, but identified by their position.

A `get` or `copy` operation specifies a template for the tuple to be retrieved. The template looks like a tuple itself, with filter values filled in for several of the fields of the tuple. This means that the system knows the tuple type to match against and has some guidance towards which tuple to return.

The design choice of tying template matching into the architecture differs from event systems, that consider filtering and matching to be services. A tighter binding between this construct and the system leads to better optimisation, but discourages different usage patterns. One possible optimisation, touched on in [Row98], is to build usage statistics. In a distributed tuple space, it would be best to hold the data closest to the place where it is going to be used. If the system 'understands' the templates, it can accumulate statistics on their usage and move the data to where it is expected to be consumed.

Linda

Linda [Gel85] is the system that first explored the idea of a tuple space. The Linda programming language provided synchronous versions of the operators `get` and `copy`. The original names for the three primitive operations on a tuple space were:

in. is an operator from the perspective of the application; in the above terminology the `in` operator would be called `get`

out. is also an operator from the perspective of the application. The above terminology would call this operator `put`.

read. again is an operator from the perspective of the application. The name given to this operation in the previous section is `copy`.

Java Spaces

Java Spaces [W+98] is a modern implementation of a tuple space system. It is part of Sun Microsystems's JINI effort. The terminology within *Java Spaces* is as follows:

read. is the term for copying an item out of a tuple space.

write. is the term for inserting an item into a tuple space.

take. is the term for extracting an item from a tuple space.

Java Spaces offers the following features beyond Linda:

Strict Typing. As opposed to structural type equality as in Linda, *Java Spaces* uses *Java* types, which are distinguished by structure and name.

Richer Types. Linda allowed for arrays, structures and base types, *Java Spaces* allows the use of any *Java* type (including inheritance, methods, collection types, etc.).

Several Spaces. As opposed to Linda, *Java Spaces* allows for the creation of multiple distinct tuple spaces. This complicates usage, since the user now must specify, exactly which tuple space to talk to (this information was implicit in Linda).

Transaction Support. *Java Spaces* allows the actions on several tuples and tuple spaces to be grouped together.

Synchronous, Asynchronous and In-Between. *Java Spaces* allows for synchronous and asynchronous matching. The synchronous matching features a timeout, allowing the user to specify a maximum waiting period.

Notification. Going beyond asynchronous `read` and `take` operators, *Java Spaces* allows a program to register interest in the appearance of a desired tuple (object, in *Java Spaces*). Upon the tuple occurrence, the server sends an event to the interested application program. The event is sent through a separate but related service in the JINI effort. During registration the client provides the server with a template for server-side filtering.

Garbage Collection. Space in one of the *Java Spaces* is *leased*, so that it gets freed after a certain time. This solves the problem of accumulating entries that are never used.

NO EVAL. The *Java Spaces* implementation does not have the equivalent of the Linda `eval` operator.

IBM *T Spaces*

The currently most advanced implementation of a tuple space system is IBM *T Spaces* [WMLF98].

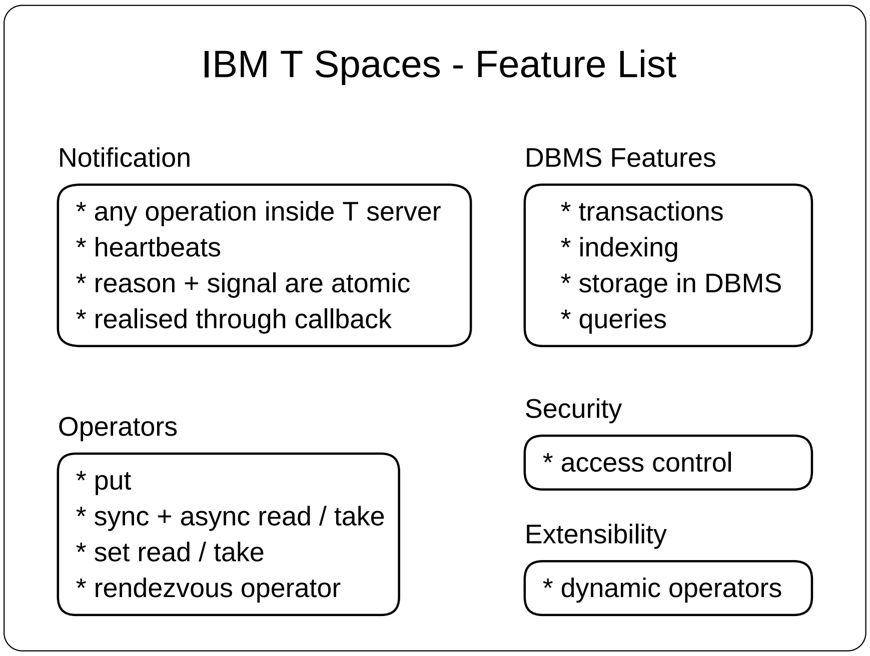

Fig. 2.4: IBM T Spaces

T Spaces is a tuple space implementation in Sun Microsystems's *Java* programming language with extensions. *T Spaces* does not communicate tuples, but *Java* objects derived from a common class. While tuples are records of base types, *Java* objects are more expressive in that they are records with members from a much richer type system and also carry method code.

While Linda provides three blocking operators named `in`, `out` and `take`, *T Spaces* additionally provides non-blocking versions of `in` and `take`. When the non-blocking operators operators are used to access the tuple space, the client operates in a mode that polls the availability of events repeatedly. This taxes the resources of all parties involved. In order to avoid this situation, the tuple space also provides notification call backs on any event occurring inside it.

The `rhonda` operator allows for a synchronous communication between two anonymous processes. Each process performs a `rhonda` operation with an argument and a type. The tuple space will match two request such that the type requested by either side matches the type of the datum supplied by the other. The return value in each case will be of the type specified in the local `rhonda` statement.

In addition to the new synchronisation modes, *T Spaces* also provides new access methods to the data inside the tuple space: the `scan` and `consumingscan` operators allow for the atomic retrieval of whole collections of tuples.

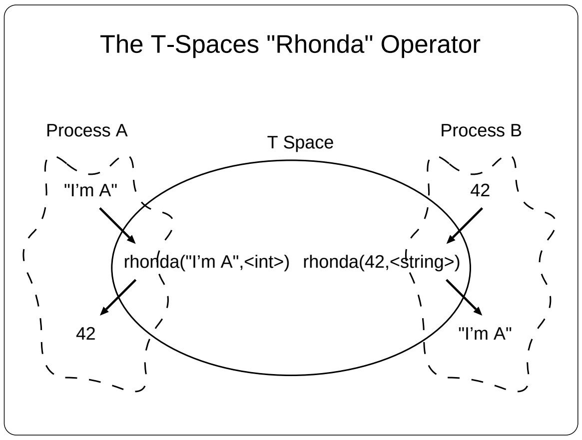

Fig. 2.5: T Spaces rhonda() operator

T Spaces provides persistence through an abstract data management layer. This layer can either utilise a light-weight in-memory database or a relational DBMS as its back-end. Either way, indexing and query facilities are available to efficiently specify and retrieve the abovementioned collections.

Given the dynamic loading capabilities of *Java*, *T Spaces* allows new operators to be defined and loaded into the servers at run-time. As a result, it is possible to define new data types at run-time and allow for evolution without shutdown.

Finally, *T Spaces* provides access control, which was not present in previous tuple space implementations. Each tuple space server may contain several tuple spaces. Access to each tuple space is granted on either user or group basis. The operations that can be controlled individually are interaction operators (in, out, take, rhonda) and the administration interface (notification registration, creation/deletion of sub-spaces).

True to the ideal of tuple spaces, *T Spaces* does not force applications to conform to a schema of available data types. While this approach clearly reduces complexity in the application development stage, it may very well increase the complexity of maintenance. The application developer has the benefit of the strong typing inside *Java*, but the system administrator is left without information as to the origin of particular pieces of data. This may make it very difficult to perform garbage collection. It seems that interoperability could be much improved through the provision of metadata.

2.2.3 Event Systems

Given the overall idea of Events, as described in 2.2, this section will present some implementations that have their roots in best effort, low latency notification.

Events, Constraints and Objects

The Events, Constraints and Objects (ECO) [SCT95] system from Trinity College, Dublin, applies event technology in the classical systems sense: event transmission is regarded as reliable and the coupling of components is considered so tight, that a global communications model is employed[5].

ECO features a run-time system with classes, objects, constraints and events as its principal concepts. *Class* is the term for an object type, the term *object* describes an object instance. An *event* – just like in Smalltalk [Sha89] – is a method invocation and a *constraint* is a boolean expression.

In ECO, an event encapsulates the parameters to a one-way method call. In other object systems, a method is invoked upon an object reference. In particular, in message passing systems, the dispatching of the message means handing it to the event message handler of a particular other object. ECO objects hand their events not to a specific other object, but to the run-time system. The run-time system is then responsible for delivering or discarding this message.

The ECO run-time system allows bindings between message types and methods to be established dynamically. This means that an unrestricted number of methods can be bound to each event type. It also means that an event type can be delivered to several methods of the same object, so that a single function call causes several invocations on the same object.

ECO is novel in that it decouples the producer and consumer of method calls, in that it allows for parallel and distributed operation and in that it allows filters (constraints) to be inserted at the producing and consuming end.

It is of particular interest that the binding of constraints to run-time instances of object types is performed dynamically. Using this technique, different instances of a single object type can be made to appear very different to an outside observer. This is a great advance in flexibility. The difference between ECO and tuple systems in that in ECO events are more strongly typed.

ECO has been applied to building large, shared virtual world models [OC96, OCDC+96], where dynamically generated constraints were used to manage the complexity of scoping.

CORBA Event Service

The *CORBA* Event Service differs from the model described above in that it allows `Proxy` objects to be inserted between the participants. The reason for those intermediaries is to allow for automatic, external composition of event flows. A `Proxy` object inherits the interface of the corresponding supplier or consumer, but extends it with a method for accepting the connector to its future communications partner.

The specification in [Obj98b] explicitly states that *CORBA* events are not *CORBA* objects, because *CORBA* objects lack identity. All the event *services* are implemented as objects. The *CORBA* Event Service document [Obj98b] provides the interface definition

[5] Other systems require the user to set up channels, before any events may be communicated; global communications permit that any participant may communicate with any other participant directly.

and a description of the underlying service. It supports `push` and `pull` interactions of typed and untyped events (as shown in figure 2.6).

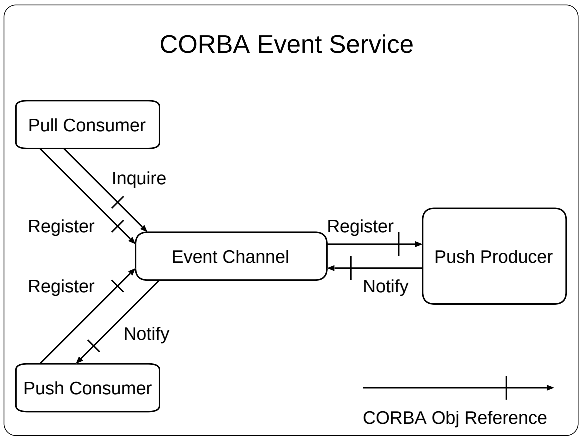

Fig. 2.6: CORBA Event Service

Push Interaction. This is the style of interaction driven by the supplier. A consumer, whose object interface conforms to the `PushConsumer` interface definition, registers interest with a supplier. Whenever an event of interest occurs, an object is generated on the supplier side and passed to the consumer through an invocation of the `push(in any data)` method of the `PushConsumer` interface.

Pull Interaction. This is a polling, consumer driven style of interaction. The consumer again registers interest with the supplier (so that the supplier will buffer events) and subsequently invokes the `any pull()` method each time that it is ready to receive the next event. It is then blocked. Besides this blocking interrogation interface, the consumer may use the `any try_pull(out boolean has_event)` interface. The supplier must buffer the events in either case. It knows to do this since a permanent connection is established before the interaction and broken down afterwards.

Decoupling. The *CORBA* Event Service defines an `EventChannel` interface, which decouples supplier and consumer. This event channel even allows for different styles (push vs. pull) to be utilised by different participants on either side of the data stream. It supports several suppliers and several consumers.

Filtering on Type. The *CORBA* Event Service allows event channels to be composed. Since a typed event channel only has the means to signal events of its own type to its consumers, it will ignore all other types of events. This is effectively a filter based on event types.

The Pull-Style interface is a practical way of interfacing with legacy systems. The system developed locally differs from this by moving the same functionality into a wrapper that provides a 'poll and buffer' interface around the legacy application, making it look like a `PushConsumer` to the rest of the infrastructure.

The *CORBA* Event Service provides untyped communications by sending *CORBA* `any` values. the *CORBA* approach to sending typed events is to agree an interface definition between the supplier and consumer that only contains methods with uni-directional behaviour (no return values and all parameters are `in` for the supplier, and `out` for the consumer).

Typed Event Channels. It is interesting to note that *CORBA* does support channels for typed events. It does this since the implementation understands the structure of a remote invocation data packet in transit. It provides all the benefits of an event channel (decoupling of supplier / consumer, interaction type of either push or pull, decoupling of supplier / consumer identity, aggregation of suppliers into one channel, distribution of events from one channel to several consumers, as well as filtering)

CORBA Notification Service

The *CORBA* Notification Service (as specified in [(Ed98]) extends the *CORBA* Event Service with the following capabilities while preserving all semantics of its predecessor:

Structured Events. Although the *CORBA* Event Service provides for structured events, the implementation and use of that service is quite tedious. The *CORBA* Notification Service improves this situation by offering an alternative mechanism for using structured events. Figure 2.7 shows a structured event as prescribed by the Notification Service. The fixed header contains a `domain` field, which is intended to be a global 'industry identifier', it contains a `type` field, which categorises the event inside the domain (this would be a term drawn from an ontology as for example defined in [AHHR+93], although the standard does not prescribe any approach to the management of these `type` field contents) and it contains the `name`, which is application provided, but intended to uniquely identify each event. These fixed header fields are followed by optional headers. Optional headers are pairs of a label and a value. The standard offers a list of optional headers that each *CORBA* Notification Service implementation must recognise (such as `EventReliability`, `Priority`, or `StartTime`). In addition, users may define their own optional header fields. The body of the message is split into two parts, each of which may be empty. The structured part contains labelled and typed data items which can be parsed by the filtering service. The unstructured part may contain any data the user wishes to transfer.

QoS Interface. The *CORBA* Notification Service provides a Quality-of-Service interface on a per-channel, per-proxy and per-event basis. The following properties can be specified with respect to QoS (but implementors may choose to extend this set): `Connection Reliability`, `Priority`, `Event Reliability`, `Maximum Batch Size`, `Ordering Policy`, `Discard Policy`, `Expiry Time` and `Pacing Interval`[6]. The standard only requires that an implementation recognises the options. The standard does not require that it supports all of the above.

Utilisation reporting. The *CORBA* Notification Service provides a `subscription_change` interface on each channel so that a server can discover which events need to be notified. Using this interface, the server does not need to construct events for which no interest has been registered.

[6] The pacing interval, together with the batch size limit specifies how sequences of events are treated, cached and forwarded.

Content Based Filtering. Each *CORBA* Notification Service channel or proxy supports the filtering interface. This interface allows a filter object to be attached at the given point in the flow graph, so as to restrict the events that are forwarded. The actual filtering expressions must conform to the *CORBA* Trader Constraint Language, with special extensions for notification. In addition to those masking filters, two special filters, called `Mapping Filters`, can be used: a `Mapping Filter` pertains either to the `Priority` or `Lifetime` field of the event and features a value associated with each filtering rule. If the rule is responsible for the forwarding of the event, its associated value is placed into the appropriate field of the event.

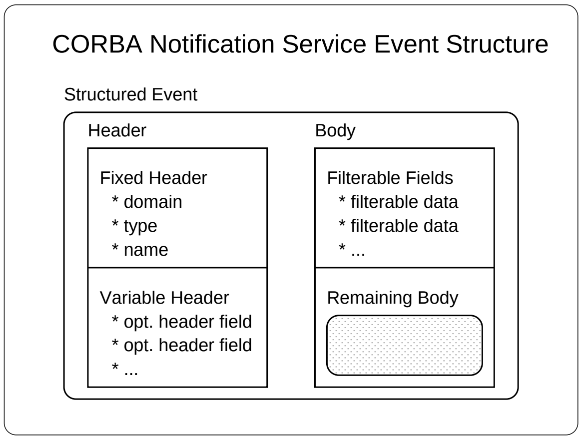

Fig. 2.7: *CORBA* Notification Service Message Structure

Optional Event Type Repository. The *CORBA* Notification Service defines a framework for structured events. In order to enable applications to adapt to new event types dynamically, it is beneficial if an Event Type Repository is present. Such a facility lists the names of event types with their associated structure.

The most significant additions of the *CORBA* Notification Service are the ability to evolve the available event types and to provide generic filtering. Quality of Service is addressed, but not defined as rigorously as the other aspects of event transmissions. An interesting aspect raised by the discussion of QoS inside this document is the fact that the term 'delivery reliability[7]' does not make sense if several decoupled consumers are to be supplied. MOM systems, which are mainly aimed at reliable delivery and mostly attempt to support exactly-once semantics, solve this problem by supplying a dead-end with each channel, so that messages are not lost, even if no consumer can be found.

The Generalised Event Monitoring Language

While the two *CORBA* systems described above receive a lot of attention and have very broad and ambitious aims, the Generalised Event Monitoring Language (GEM) in [MSS95] has a much more narrow focus. GEM is based on the REGIS [MDK94a] / DARWIN [MDK94b] development environment. REGIS provides an infrastructure, in which programs are constructed by specifying how various building blocks are configured. The configuration language is called DARWIN.

Where *CORBA* employs filtering only, GEM allows for the rule-based selection of events from a stream, their de-construction and the composition of a new event from those fragments. The area closest to GEM's approach is that of active databases. GEM

[7] Delivery reliability is normally categorised as *best-effort*, *at least once*, *at most once* or *exactly once*.

differs from those systems in that it has to operate in a distributed environment, while most active databases are centralised. Due to this distributed nature, new problems arise:

Parallelism. In a centralised world, time ordering can be achieved reliably. In a distributed world, an order in time can sometimes not be established with certainty, because either the clocks are not comparable[8], or the events have actually occurred simultaneously.

Network Latency. Even if the time ordering between two events could be established reliably, it may still be the case that they arrive at the location where the comparison is to be carried out in the wrong order. GEM allows a time-out to be specified before the event is declared as non-existent and the expression is evaluated respectively.

Timestamp Manipulation. When a composite event is constructed, it could either be assigned one of its component timestamps or it could be given its own timestamp during its construction. GEM only allows for two special cases: the construction timestamp, which is called `notify` or the faithful replication of a received event, called `forward`.

While GEM tackles all of the above, it did not address issues like the consumption policy for events, resource discovery, access control, federation and evolution.

A Cambridge Event Architecture

Development of the Cambridge Event Architecture (CEA) commenced before the *CORBA* Notification Service was published, and addresses some of the same problems:

- CEA supports the *publish-register-notify*[9] model for direct (client / server) or indirect (client / channel(s) / server) communications.
- CEA features filtering at any node inside the transportation path for events. CEA can filter on type and also use a very expressive composite event detection language, which is based on a sound theoretical foundation [Fer97]. The language supports the operators `without`, `sequence`, `or` and `whenever`. An `and` operator would imply simultaneous occurrence, which is difficult to check in a distributed system, but users may use the expression $(a;b)|(b;a)$ to express that they expect both occurrences in arbitrary order.

[8] GEM deliberately ignores this problem by assuming a global clock and tagging events at the first stop that does possess a clock. GEM also records the source, which means that a comparison could theoretically be performed if the skew at the sending time between the two clocks is known.

[9] While many systems are described by the term *publish-subscribe*, CEA uses the terminology *publish-register* to emphasise its more finely grained model [BBHM95]. In CEA clients do not subscribe to a large stream of events, filtering out useful information, but they register interest with particular sets of events with the next upstream node.

- The CEA architecture has been designed to inter-work with role-based access control provided by OASIS [Hay96].
- CEA supports fine grained service levels, in that events are batched for transmission and the timing can be adjusted arbitrarily. Reliability is added through a heartbeat protocol, which operates at the same parameterised granularity, giving not only control over the tradeoff between latency and link utilisation, but also providing the connection status.

CEA and GEM are similar in that they focus on composite event detection. But while GEM is a system, CEA is a framework. An illustrative is GEM's builtin timer, where CEA expects such information from the outside. GEM is a rule-based system, similar in operation to expert systems [BFKM85] or an active database [MD89]. Rule-based systems always need a very careful analysis to determine the number and order of invocations caused by a single event. In CEA this difficulty is avoided through structuring: each detection stream is consumed by exactly one expression. Both GEM and CEA differ from the *CORBA* Notification Service in the fact that they cannot only filter, but actually construct new, composite events.

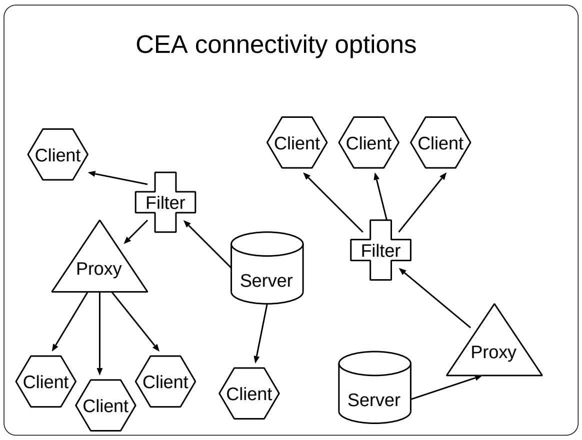

Fig. 2.8: CEA Connectivity

Microsoft *Active Channels*

This last event system is very different from the examples above, because it operates on top of a much more complex protocol stack and it is very much more specific in its purpose. Finally it is an excellent example of the dilemma faced by event systems.

Microsoft *Active Channels* [Net97] are implemented on top of the Hyper Text Transfer Protocol (HTTP). The system consists of a definition file format to define the channel and an interpreter for channel definition files at the client end.

This implementation means that the illusion of asynchronous server-driven notification is actually achieved through periodically polling by clients. The definition files contain the location and frequency of the content to fetch.

This approach has the advantage that the server is state-less. With millions of arbitrary Internet users, a server cannot provide permanent resources to establish a channel and retain the connection information. It also has the advantage of leveraging existing infrastructure. A wide network of WWW-caching servers is in operation around the world. They form a distribution hierarchy and as a result only a very small percentage of the users' update requests should reach the main server. Most of the update requests should be serviced by the caching servers, which themselves honour expiry information.

Microsoft *Active Channels* is a special purpose solution that provides the illusion of asynchronous delivery to the end-user. It uses existing protocols and infrastructure to ensure scalability.

Even though the *Active Channels* are based on a sound design, the technology has not been taken up. There are at least two possible explanations for the failure of this technology: (1) it is a special-purpose, end-user application for a service that the end users did not want and (2) it has not been used to its full potential. While it was designed to leverage the proxy infrastructure of the WWW, company lawyers frequently enforce the condition that content be marked non-cacheable. This is to retain ownership of the content. Without the use of caching servers, the main providing servers were overwhelmed and unreachable.

2.3 Directories, Traders, and Schema Repositories

This section surveys different approaches to metadata storage and provision. The historical terms for such services are directory systems, schema repositories and traders. Directories tend to hold mappings from human readable data to machine internal data, schema repositories describe the relationships of data inside a database management system and traders list alternative services and their attributes.

These systems all maintain records of well-defined structure, but they differ in the service levels and convenience functions they provide. Service levels might be access control, authentication and transactional updates, where convenience functions might be translation of character sets or searching of sub-trees. The metadata stores described in this section differ in the following ways:

Consistency. The spectrum spans from a database schema, in which it is impossible to ever observe inconsistency between the schema and the database tables to an address book, in which each entry is entered by hand and the system has no automated way of verifying the correlation of address and name.

Performance. A repository can be optimised for various access patterns, such as search for specific attributes, or inexpensive updates. The most frequently employed optimisation currently lies in repositories for replaying of scenarios involving continuous media where ordering is performed on the timestamps.

Reliability. The data in the repository may be protected against system failures, media failures or even tampering.

Scalability. How many users can be supported by the system, considering that users need not be humans but can be automatic entities.

Manageability. Whether it is possible to delegate control of separate parts of a directory to separate entities.

Data Structuring. Data inside the repository may be structured as a set, as a collection of sets, or maybe even as a graph with or without restrictions.

Access Control. Different policies may be employed to restrict access. Not everybody may be allowed to perform update operations, and even with public information, quantitative limits may be imposed on the amount of data retrievable from the system, in order to obstruct outright copying.

Query Processing. Query access to the repository may be provided at differing levels of sophistication: a directory or trader supports fuzzy matching, while a database schema only supports direct lookup.

The following sections will portray several example systems, highlighting design choices and filling in historical developments.

2.3.1 The Internet Domain Name System

The 1983 specification of the Internet Domain Name System [Moc83] comprises three concepts: *domain name space*, *name servers* and *resolvers*.

The domain name space is a concept for the distributed construction of mutually exclusive names. It has the form of a tree with labels attached to each edge. The labels of all edges emanating directly from a particular vertex are under the control of the same management entity. The unique name of a particular domain is formed by concatenating from left to right the names found on the path from the most specific vertex to the root of the tree, using dots as separators.

Each name server is an active process that can answer queries about the domain name space. Name servers can retain information about any part of the domain name space, but typically they are assigned the management of a particular part of the domain name space and can answer queries about this part with authority. That part is called their *zone*. The same process may still cache information about other parts of the domain name space, but without authority over it.

Finally, a resolver is a program that interacts with the server processes to perform a distributed query on behalf of some user.

Having created such an infrastructure of unique names, information can now be attached to each of these names. One particular piece of information that can be attached to a vertex is different from the remaining forms: an alias[10] designates the vertex as

[10] This type of record is called CNAME. CNAME stands for 'canonical name', which means that the node in the tree is an alternative name for the items actually located at the location given in the CNAME record.

an alternative name for another vertex.

Besides this meta information, it is possible to attach an arbitrary number of data records to each vertex. These data records can be of various types, with more than one record of each type allowed to coexist. The most important record types are:

A. A host address, meaning that the name of the vertex describes a host and that this host can be reached via the address given in this record.

NS. An authoritative name server, meaning that the part of the tree cached at the local server delegates management of a sub tree to a separate name server. The name of that server is given inside the NS record.

MX. A mailbox record contains the host name of the machine to which mail addressed to this node in the domain name space should be sent.

HINFO. A host information record asserts that this node in the domain name spaces describes a host computer and the record contains information about the hardware and operating system of that machine.

RP. A record naming the responsible person for this particular domain or host contains the electronic mail address and the name of a descriptive TXT record. RP records are used to notify the management of the name server of either general DNS problems or specific security incidents.

TXT. Text records may contain arbitrary, user defined textual data. They are provided as a means of keeping the system extensible.

The domain name system has a number of predefined record types with associated semantics. This means that all software accessing the DNS is expected to conform to those minimum standards. The types and their meaning form a schema, with structural and semantic specifications provided in English language form. Almost ten years later, a further recommendation (see [Ros93]) was published on how the extension mechanism of the TXT field ought to be used to provide a general directory service. This purely syntactic convention suggests that data be provided in the form of tuples, giving an attribute name and value, which are separated by an equals sign. The recommendation explicitly avoids addressing the management of the attribute names and their interpretation.

2.3.2 Sun Microsystems *NIS+*

NIS+ is the successor to the Network Information Service (NIS). The data in this directory is structured as a tree. Each node in the tree is called an *NIS+* object. Each *NIS+* object has one of the following types: directory, entry, group, link, table and private. *NIS+* differs from NIS by providing data encryption and authentication.

Structurally, *NIS+* provides the same capabilities as the DNS, but it adds simplified searching, it operates over RPC rather than through a stream based protocol and it has an authenticated update interface.

The *NIS+* root object is always a directory. A directory can contain directories, groups, tables, links and private objects. This is similar to the DNS, where a list of resource records (a table in *NIS+*) can be attached to each name in the name-space. *NIS+* links are similar to 'canonical name' resource records in DNS.

Where DNS defines record types, *NIS+* offers guidelines for the structure of each name space: the root directory of a local name space in *NIS+* contains the directories `org_dir` and `group_dir`, which contain information about the users, hosts and groups inside the domain.

2.3.3 The international X.500 electronic directory standard

Five years after publication of the DNS specification, the International Telecommunications Union (ITU) published the *X.500* standard [X.598] with the title 'Data Communication Networks Directory'.

The directory system described in the recommendations *X.500* through *X.521* is a very general and high level system for maintaining a graph of structured names. Initially, the directory is assumed to be a tree, but the specification explicitly states (*X.500* §6, page 7) that future uses of the *X.500* standard may allow other topologies.

The information contained in the directory is called the Directory Information Base (DIB). The graph of names is called the Directory Information Tree (DIT). Each entry in the graph has a *distinguished name* derived from the hierarchy, making it unique and unambiguous.

Vertices in the DIT are called entries. Each entry is either an object or an alias with exception of the root, which is the special case of a null object. Each entry has a type, with alias types restricted to leaf entries. As each entry has a distinguished name, alias entries are named references to other vertices in the DIT. The *X.500* specification allows alias entries to be returned to a user agent in un-resolved form. This increases the flexibility of the system, but puts a greater burden on the client. While the *X.500* directory appears to operate just like a UNIX file system directory, it does not allow for relative addressing. All names must be quoted in their absolute, unabridged form.

The structure of objects and aliases is the same. Namely, they both contain attributes, with each attribute giving its type and one or more values from the domain that its type prescribes (see fig. 2.9). The attribute type is an object identifier and each entry can only contain zero or one attributes of any particular type.

The Directory has a set of constraints that specify the integrity conditions for the DIB. These constraints are called the *Directory Schema*. The Directory Schema restricts

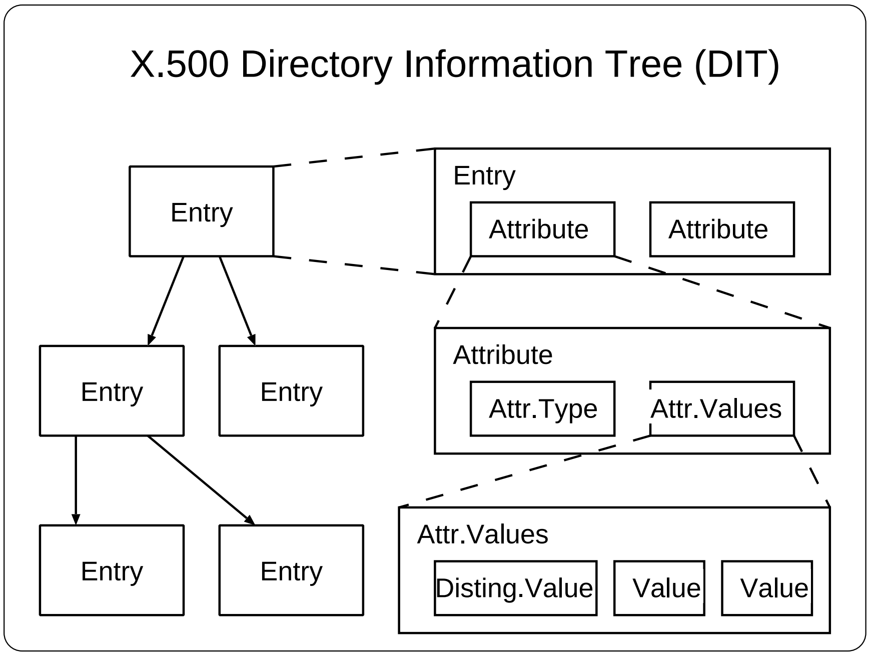

Fig. 2.9: X.500 structural overview

which object types may be related inside the DIT, it contains the object class definitions with allowed and required attributes, it contains attribute type definitions with its identifier and whether it may be single or multi-valued. The DIT contains attribute syntax definitions which describe the structure of the type, along with matching rules (for example 'sound-alike' matching) for that type. The Directory Schema is stored in a distributed manner, with each authority storing the part of the schema that applies to the part of the DIT that it is managing. While the 1988 version of the standard does not offer an interface for schema update, such an interface is supplied in the 1993 edition. Without an interface to manipulate the schema, the 1988 edition of the standard does not even specify how the schema is stored inside the directory.

The 1988 version of an *X.500* directory service provides the following services:

- The directory can be used to look up some or all the attributes associated with a particular name.
- The directory can be browsed to see which names are used.
- The directory can be searched by attribute value, yielding as a result the set of names inside the directory representing the objects whose properties conform to the requested attribute values.
- Access control at a fine grained level (the following access categories may be protected: detect, compare, read, search, modify, add/delete and naming). An important aspect is the granularity of read access: many users may wish to obtain

a complete copy of the directory for purposes not acceptable under the stated terms of use. Thus, it may be necessary to limit the number of answers returned per time period and per region. Authentication in *X.500* is performed through the Kerberos [SNS88] protocol.

The major additions of the 1993 edition of the *X.500* standard are concerned with performance issues (replication management), access control management, schema management and searching mechanisms.

X.500 is a very detailed definition of a very flexible directory system. It depends on the large infrastructure of the *Open System Interconnect* (*OSI*) reference model, which is a deterrent to its implementation.

The next section on *LDAP* will show how to take advantage of the *X.500* flexibility without incurring its resource costs.

2.3.4 Light-Weight Directory Access Protocol (LDAP)

The *Light-Weight Directory Access Protocol* (*LDAP*) was invented by the Internet community in order to get the benefit of all *X.500* advantages without incurring the large cost associated with implementing the OSI protocol stack.

The *LDAP* operates over the Transmission Control Protocol (TCP) only. *LDAP* simplifies the access model in such a way that all work is performed by the server. While *X.500* may return an alias entry to the client, *LDAP* requires that alias resolution is performed by the server. *LDAP* supports the following operations: bind, unbind, search, modify, add, delete, compare, abandon (the abandon operation cancels an outstanding request).

The *LDAP* is important, in that it allows the advanced directory technology of the *X.500* standard to be used in desktop enterprise systems, giving it a much wider exposure than it previously had. It was exclusively used in the telecommunications industry, but *LDAP* increases its affordability and allows it to be put to use for a much greater variety of tasks.

2.3.5 Novell Directory Services

The *Novell Directory Services* (*NDS*) software is an implementation of the *X.500* directory model, with exception of the *X.500* communications protocols. *NDS* implements a set of much more light-weight protocols, suitable for Personal Computer Desktop environments. *NDS* goes beyond providing the directory database, it provides the interfaces and 'glue' software to integrate with various networked resources (such as printers, file servers, application servers, databases, etc.).

NDS keeps a complete distributed (and replicated) tree of users, resources, authentication information and constraints. It provides centralised management tools with the ability for delegation of management to other entities. *NDS* then formats part of its content in a suitable form for import into the authentication and management mechanisms of the networked resources and provides those with the derived information. The *NDS* auxiliary programs are also capable of detecting discrepancies between the management information in a resource manager, like a file server, and update the central directory to match the new data. One of the major aims of *NDS* is to provide 'single login', which means that a network user authenticates against the *NDS* only and *NDS* performs on-demand authentication against all other networked resources. This alleviates the user from the burden of managing several passwords, thereby increasing security.

Hence *NDS* is not only a directory service, but a suite of network management tools centered around an *X.500* structured directory system. Finally, *NDS* also supports the *LDAP* protocol, making it suitable for inter-operation with an even wider class of applications.

2.3.6 Microsoft *Active Directory*

Microsoft *Active Directory* will combine several of the above technologies. It will locally provide the the functionality of a tree structured name graph with an *LDAP* interface. It will, however, recognise that the global infrastructure of the Internet Domain Name System is much more mature than the international *X.500* infrastructure. In accord with this observation, it will allow DNS leaf nodes to be *Active Directory* root nodes. This combines the detailed specification of *X.500* fine grained attributes with the wide deployment of the DNS. Being based on *LDAP*, it inherits all of *LDAP*'s properties.

On the implementation side, *Active Directory* will feature replication, which the documentation calls 'multi master', it will also feature an indexing engine that allows for aggregation and fast searching of the content from several directory instances.

Active Directory will feature supporting software with several authentication protocols (including 'simple' and 'Kerberos'), transitive trust relationships, management delegation and software deployment to machines represented by entries within the directory. This technology was originally scheduled to be included into the first launch of *Windows 2000*, but has now been postponed to about half a year after its launch.

2.3.7 *CORBA COS Naming*

The *CORBA* Common Object Services (COS) Naming service differs from the above approaches in several ways:

Topology. The COS Naming service allows for arbitrary directed graphs to be constructed.

Attributes. Each node in the graph contains exactly two attributes: the partial name and an object reference. Additional attributes about the the object can be maintained with respect to the object reference through an instance of the Association Service (see [Obj98a]). While it may not be possible to find out the structure of the object at all, it will always be possible to find out the structure of the interface by handing the object reference to the *CORBA* interface repository.

Addressing Mode. COS Naming service names are always interpreted relative to a particular naming service. No global authority exists. A client using the COS Naming service must have a reference to a specific context object. Attempts to resolve a name inside the wrong context lead to undefined results.

The COS Naming service defines a `NamingContext` interface. An object implementing this interface maintains a set of names with associated object references. A naming graph can be created by binding the object reference to a second context to a name within a first context.

The COS Naming service specification provides a standard representation for a human readable form of relative name sequences. The structure of the service allows for most other naming schemes to be mapped into and out of the COS Naming service without loss of information.

These two properties make the above proposal a very open solution. Text is a very basic form of data and supported on all platforms, and the chosen directed graph model for naming can subsume all naming systems in widespread use. As a result, tools written for the above model will be able to interface with a great variety of systems.

2.3.8 X/Open Federated Naming

The X/Open Federated Naming system (XFN) (see [Mar95]) is part of the Distributed Computing Environment (see 2.1.4). It is a set of interface specifications for performing a lossless mapping between different naming systems. A *DCE* installation may need access to names from

CDS. the *DCE Cell Directory Service* (a free-form directory with very fine-grained access control)

GDS. the *DCE Global Directory Service* (an implementation of *X.500*)

DFS. the *DCE Distributed File System* name space

DNS. the Internet Domain Name System

NDS. the *Novell Directory Services*

XDS. the *DCE X.500* context implementation, which is a client to some *X.500* directory server

NIS+. the Sun *NIS+* directory system

Security. the *DCE* Security Service which names principals for access control, such as users, groups, computers and processes

Furthermore is it envisaged that a *DCE* application has applications that implement name spaces that are separate and proprietary again. Examples of such applications would be databases, desk-tops and spreadsheets.

The XFN allows for seamless access to these federated[11] name spaces. Due to the diverse nature of the underlying name management systems, advanced features (like constraints or schema management) are not supported by XFN.

2.4 Transaction Services

Programmers have two quite simplistic, but for most applications sufficient, models for failure. Namely, they expect the machine to halt on failure and they expect that all volatile memory is lost. The second expectation concerns media failures and assumes that media loss is dealt with by replacement of the logical device with a previous snapshot of the state.

Distributed systems introduce the possibility for the failure of individual components. If at least one of the end points of the communication is a human, as in electronic mail, chat sessions or a web browser, then it is easy enough to assume that this human can deal with complex failures by recovering components independently to a consistent state.

For complex distributed systems, involving several machines and with a need to reason about the correctness of the computation, the more formal approach of *Transaction Processing* has been developed.

Transactions have originally been applied to database management systems and ensure four properties for a given computation:

Atomicity. If the transaction encloses several operations, then the atomicity property ensures that either the effects of all operations or none of the operations are observable. Atomicity ties together the operations into a single unit that can not be applied in part only.

Consistency. A database management system allows the specification of *constraints*. A constraint is a logical assertion that must hold true for the observable part of the database at all times. An example of such a constraint would be that only orders with valid customer numbers may be stored in the database. A customer number is only valid if a record giving the customer address exists that describes

[11] 'Federated' in this context means independent, but co-operating.

this customer number. This is a mechanically confirmable condition and therefore can be used as a constraint. Consistency is checked after all operations enclosed by the transaction have been applied. If the check succeeds, the results are made visible, otherwise the whole computation is discarded and the database is left untouched.

Isolation. Database management systems restrict transaction programs to perform operations in such an order that *serialisability*[12] is achieved. From the perspective of an individual transaction program, this property creates the impression of completely isolated operation. Since database values read by the transaction are from a consistent state, it seems as if no other transaction is concurrently in progress.

Durability. Database transactions also provide the guarantee[13] that results are not lost once the confirmation of successful transaction completion has been issued. This choice of words describes an analysis of probabilities, which pits the failure rate of redundant disk drives against that of main memory corruption. The design of current transaction systems contains failure analysis of the individual components and balances the failure rates of processors, memory and communication links against the failure rates of redundant disc and tape storage systems.

Transactions of this style are called 'flat transactions'. An important extension of this model is that of 'nested transactions'. As described in [BM91], nested transactions allow the creation of sub-transactions within parent transactions. Results of sub-transactions are only visible inside the parent transaction, but not outside it. This means that a rollback of the parent transaction implicitly rolls back the already concluded sub-transaction. The use of this feature lies in much finer granularity for error containment. An even more generalised model for transaction dependence can be found in [BDG+94], where commit and abort dependencies of transactions can be specified in an arbitrary graph and synchronisation primitives are provided to allow the passing of resource responsibility among the group.

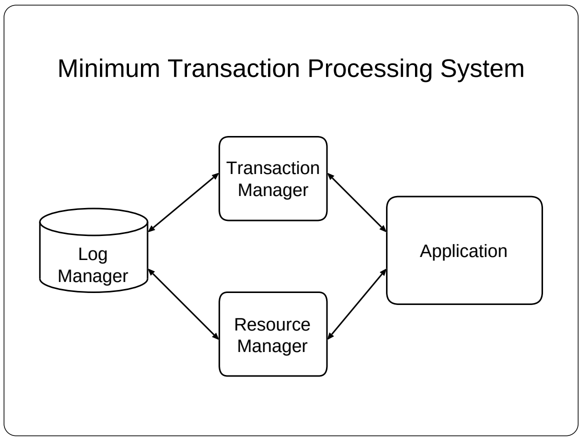

Fig. 2.10: Minimum Transaction Processing System

Transaction systems for distributed computing ensure the same properties, but *consistency* is not typically as configurable as in the database world.

A minimum transaction processing system (as shown in figure 2.10) contains an application, a transaction manager, a log manager and a resource manager.

[12] Serialisability is the property that a sequence exists in which the transactions can be applied serially to take the state of the database through the same transformations as observed with the present, concurrent execution.

[13] The largest database vendors are located in the United States of America, where the word 'guarantee' is not used for legal reasons. Those vendors use the term 'assurance'.

A minimum transactional application operates in the following way: it requests a new transaction identifier from the transaction manager. The transaction manager generates the requested data and records the fact in the log before returning control to the application. For each operation that the application wishes to have performed by the resource manager, it passes the transaction identifier along. The resource manager only supports reversible, idempotent[14] operations. It records each operation, together with the necessary undo- and redo-information in the log. Finally, the application requests the transaction manager to commit the transaction. As a result, the transaction manager first records the request in the log, then inquires with the resource manager whether there have been problems and finally records successful completion in the log.

With this model, the four properties of transactions are achieved as follows:

- Atomicity could break by having a program begin, record changes and then stop without committing the changes. In this case the transaction manager (either on timeout or restart) traverses the log backwards, searching for records pertaining to the given transaction. Each log record is handed to the resource manager for un-doing. Eventually, all effects of the application have been reversed and no changes are left. Another way to break atomicity would be to lose part of the changes to a resource manager. This could happen through media loss or power failure following commit. In this case, the log again contains all information necessary to remedy the problem. All changes are re-applied and no data loss is noticeable.
- Consistency is defined at application level. In the transaction system described above, the consistency logic is encapsulated by the resource manager and the resource manager gets a chance to veto the commit operation just before it happens. Another option would be for the resource manager to fix consistency violations itself (an example would be to update indices after changes have been made to a data structure).
- Isolation is maintained by a set of mechanisms called 'concurrency control'. One could ensure isolation by ensuring that no conflicting applications are executed concurrently[15]. These preventive measures are termed 'pessimistic concurrency control', because they expect a conflict. 'Optimistic concurrency control' simply records the resource usage and checks for conflicts at commit time, possibly aborting one or more transactions.
- Durability cannot be guaranteed, since there is always the possibility of total destruction by disaster. Durability is approached through software engineering to

[14] The word 'idempotent' means that repeated application of the operation yields the same result; an example of an idempotent operations would be $a := 5$ whereas an example of a non-idempotent operation would be $a := a + 5$.

[15] Prevention of conflict could be done by static analysis of the required resources before invocation, or by dynamic analysis of the required resources (locking) at runtime.

reduce the risk of loss inside the machine and logistics management to reduce the risk of loss of physical media. The transaction system is engineered with a very careful attention to write order and high redundancy. The logistics for placement of the physical media are chosen so as to minimise the risk of simultaneous damage to all copies.

Finally, there is great scope for variation in the choice of components, physical placement of those components, concurrency control protocols, commit protocols and of course variation by omission, relaxation or extension of some of the above properties.

In their book "Transaction Processing – Concepts and Techniques" [GR93], on which this introduction is closely based, Gray and Reuter list a number of transaction processing systems based on their historic relevance, commercial success and elegance of design. The systems described extensively are IMS/TM, CICS, Guardian 90, DECdta and X/Open DTP. IMS makes extensive use of operating system facilities and memory protection, while CICS provides replacements for most calls and allows client code to run in the same protection domain. Guardian is a good illustration of extensive replication and fail over approaches, and it is different in encouraging the use of its transaction services primarily through one language, an interpreted COBOL. DECdta emphasises distribution and interworking. Its components are much less tightly coupled than in the other systems. X/Open DTP specifies an open protocol for transactional interworking.

While Gray and Reuter have undoubtedly selected the most prominent examples, the following sections will describe transaction processing standards and implementations that are of particular relevance to Middleware.

2.4.1 High Performance Transaction Managers

BEA Systems' *Tuxedo* [BS96] and Transarc's *ENCINA* [Mac98] are two state-of-the-art distributed transaction manager solutions. They provide complete distributed transaction processing environments and a review of transaction processing would be incomplete without their mention. With transaction processing being their main focus, they both provide complex transaction dependency models (including nested transactions), sophisticated locking and scheduling, management of server pools with load balancing, extensive administration facilities and support for a variety of platforms.

2.4.2 X/Open Distributed Transaction Processing (DTP)

The International Standards Organisation (ISO) has published two standards (for OSI-TP, see [Org92], for OSI-CCR, see [Org89]) for Distributed Transaction Processing. Together, these standards define a scheme for producing globally unique transaction names and the message formats for standard Two-Phase Commit (2PC) interaction.

The OSI-TP standard describes the following functional modules (as shown in figure 2.11) for a transaction processing system:

The *Application* is the user-supplied part of the program. It encapsulates the logic to be performed.

The *Transaction Manager* ensures the ACI properties of the transaction by coordinating the work-flow.

The *Resource Managers* ensure Durability and participate in the 2PC protocol by voting on consistency.

The *Communications Manager* is a special resource manager that allows access to remote resource managers and the control of a remote transaction manager.

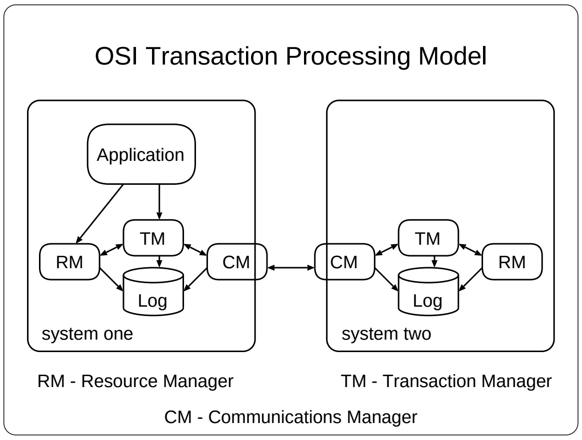

Fig. 2.11: OSI TP Model

TM	XA	XA+
tminit()	xa_open()	this is
tmopen()	xa_close()	the same
tmbegin()	xa_start()	as the XA
tmcommit()	xa_end()	interface
tmsync()	xa_prepare()	but it
tmterm()	xa_commit()	also
	xa_rollback()	contains
	xa_complete()	mgmt.
	xa_recover()	functions
	xa_forget()	

Fig. 2.12: The three X/Open Interfaces

The three Application Program Interfaces (API) from the Open Group (see 2.1.3) encapsulate these conventions [Gro91].

The TM interface is used for communication between the application and the transaction manager.

The XA[16] interface specifies how the transaction manager interacts with the resource managers.

The XA+ interface specifies the interactions of the transaction manager and the communications manager.

2.4.3 Transaction Internet Protocol

The Transaction Internet Protocol (TIP), as described in [LEK98] provides a very simple means for two parties to agree on the outcome of a transaction.

To this end, TIP defines transaction identifiers, transaction manager identifiers and resource identifiers (which are called Uniform Resource Locators (URL)), all of which are human readable.

The TIP operates on two entities, connections (reliable octet streams) and transactions. It defines a number of human readable commands (`identify`, `multiplex` and `tls` for

[16] As mentioned in 2.2.1, 'XA' is a term from the X/Open transaction processing standard and stands for 'Cross Memory Access', as opposed to a remote procedure call.

connections; and `abort`, `begin`, `commit`, `prepare`, `pull`, `push`, `query` and `reconnect` for transactions), that can be sent over the connection.

Finally, TIP maintains a finite deterministic automaton on each end of a connection, which is always in one of the allowed connection states (initial, idle, begun, enlisted, prepared, multiplexing, tls, error).

The remainder of the document deals with security considerations. Since TIP is an Internet protocol, it has maximum exposure to malicious users and therefore several scenarios for spoofing or denial of service attacks are investigated, and solutions proposed.

To summarise, TIP is of great interest, since it covers an extremely small, but well defined problem out of a very large problem domain. Its design choices are in accord with other Internet standards, in that data is human readable when it can be, that a minimum implementation effort is needed and finally, that commonly occurring problems in similar protocols are avoided in TIP.

2.4.4 CORBA Object Transaction Service (OTS)

The *CORBA* Object Transaction Service (OTS), as defined in [Obj97], supports nested transactions, is based on the X/Open DTP, but differs in its classification from other systems. It identifies the following components:

Transactional Client. This would be called an application program in other systems.

Recoverable Object. Other systems would label this as a resource manager. The recoverable object has an associated 'resource object' that handles all call-back invocations from the transaction manager. It is this resource object that is registered with the transaction manager.

Transactional Object. This is an entity not found in other systems. It can request the rollback of a transaction, but does not participate in the commit processing. If it implements persistent state, it does so through use of a recoverable object. Each recoverable object is by definition also a transactional object.

Transactional Server. This is a collection of transactional objects.

Recoverable Server. This is a collection of recoverable objects.

One notable property of the OTS is that it modifies the runtime environment of the executing program: the transaction context is associated with the thread current at transaction creation time. Method calls to transactional objects implicitly pass the transaction context. This effectively means that the thread package must support the attachment of the transaction context. Furthermore the RPC mechanism must pass the implicit transaction context to the server object. The designers of the OTS have paid careful attention to the idea that the user should not need to modify their programs, but in the process they have burdened the Object Request Broker (ORB) implementer with modifying the RPC protocol. As a result, every *CORBA* method invocation must carry along enough information to allow for the transaction protocols to be executed successfully.

2.4.5 *Enterprise JavaBeans*

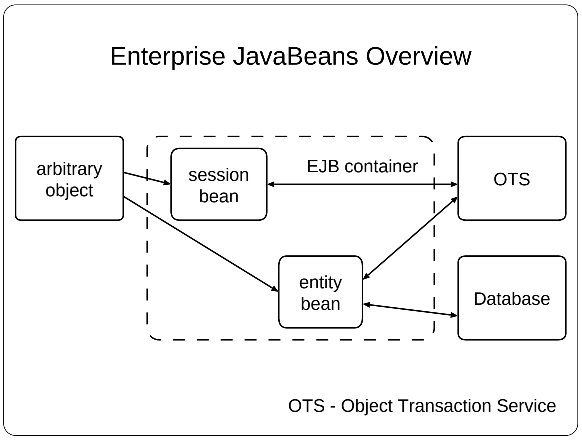

Fig. 2.13: Enterprise JavaBeans Overview

The *Java* programming language defines a virtual machine with many service interfaces. *Enterprise JavaBeans* is a second such specification, that defines a virtual machine with interfaces that are relevant in a back-end server. *Enterprise JavaBeans* is an attempt to represent all functionality of an On-Line Transaction Processing (OLTP) system in the context of such a well defined interface. This technology aims to standardise the way in which *Java* applications access such products as *IBM TXSeries* or *BEA Tuxedo*. This allows the proven solutions with very high performance to be used in a *Java* application server.

As described in [Tho98], *Enterprise JavaBeans* is a middleware solution: it defines interfaces for much of the functionality described in the introduction of chapter 2, but does not provide implementations. Specifically supported are

- *Java* Naming and Directory Services (JNDI)
- *Java* Transaction Service (JTS, based on OTS)
- *Java* Transaction API (JTA)
- *Java* Database Connectivity (JDBC)

JNDI provides a uniform interface to all directory systems mentioned in 2.3. JTS provides a wrapper for external transaction managers, for use by the EJB infrastructure (this is similar to the XA interface between a manager and the resources). JTA is an application interface to the transaction manager (similar to the TM interface). JDBC provides a uniform interface to database access and will be described in 2.5.3.

Besides this explicit infrastructure, EJB defines a component model. Figure 2.13 shows an example component. In EJB terminology a *component* is the object with state, behaviour and identity, and the during method invocation the EJB environment (the *application server* provides a *container* to manage the execution of that *component*). The server infrastructure does something similar to an operating system swapping a process back into memory: it provides a number of implicit services (life cycle, state management/swapping, security, transactions, and more specific persistence). it allocates a process thread and initiates the component. The *container* is like an advanced thread: besides memory management, resource access and compute cycles, it provides transaction and security services. The *container* manages all interaction between the *component* and the outside world.

The transaction model used by EJBs is that of flat transactions, utilising the *CORBA* OTS.

The implicit transaction management of EJBs operates declaratively: instead of a programmer inserting function calls to the transaction manager, each EJB object contains a transaction attribute, that describes a 'template' for transaction creation. The attribute could require a new transaction to be started on each invocation of a method on the object (`TX_REQUIRES_NEW`), or that the method must be executed in the context of a transaction (`TX_MANDATORY`, where `TX_REQUIRED` creates one if it doesn't exist). The imperative way of explicitly inserting calls to the transaction manager is still provided through the JTA.

2.4.6 Microsoft Transaction Server

Microsoft has integrated a further service into the *Windows NT* operating system: the *Microsoft Transaction Server* (*MTS*).

MTS is a transaction manager that provides coordination for flat transactions on DCOM (see 2.1.4) objects. *MTS* exports a typical transaction manager interface, but it also supports implicit transactions, as described above for Enterprise Java Beans.

The interesting difference between *EJB* and *MTS* lies in their approaches to modularisation:

Microsoft provides an operating system with integrated transaction services that are accessed through the binary object interface specification. On top of this binary object interface, various programming languages may be supported.

Sun provides a single programming language, but makes many external services available through well designed interfaces.

The first approach leads to tighter integration between different components, while the second approach is a true middleware solution, integrating the services and resources from many different platforms.

2.5 Database Connectivity

Middleware for database connectivity operates much like other middleware: it provides a unifying layer of abstraction from several differing database management systems. This section will first describe the typical use of database middleware in three-tiered systems and then the various aspects that it provides.

2.5.1 ODBC

The most common use of database middleware is to connect an application program to a database back end. Microsoft has defined Open Database Connectivity (ODBC) interface [Net].

ODBC is the interface definition of an API to access database management systems. Various vendors supply implementations of library files and application developers use the function calls to perform database operations. The ODBC API is a superset of the Data Management standard from the Open Group [C4496]. It implements the functionality of the *SQL* call-level interface.

SQL covers the functionalities of

- a Data Definition Language (through commands like `CREATE TABLE`)
- a Data Query Language (through commands like `SELECT`), and
- a Data Manipulation Language (through commands like `INSERT`)

ODBC has been so successful in the domain of Microsoft *Windows*, that the interface has been ported to other architectures and is now widely used to access relational database management systems.

The Microsoft ODBC specification contains three sets of function calls, denoted Level 1 through Level 3. Lower levels are subsets of higher levels and in this way three degrees of compliance are defined.

Application designers can trade off performance (through use of specialised features) against portability (more ODBC drivers with lower compliance levels exist).

ODBC is a very commercial phenomenon, but its widespread use makes it the *de facto* standard for relational database access.

2.5.2 Three Tier Systems

Designers of business applications have long been faced with the demand for reliable, cost-effective, and flexible systems. The structure of 'three-tiered systems' as shown in diagram 2.14 appears to meet that aim. These systems consist of a database management system as tier one, the application logic in tier two and the presentation part in tier three.

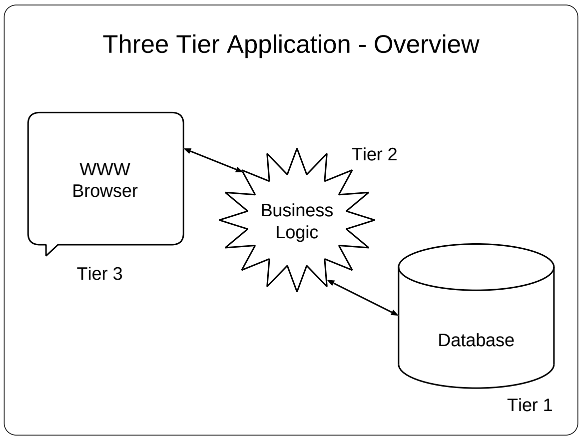

Fig. 2.14: Three Tier Application

Database management systems are very mature and provide the reliability that the business owner demands. The application logic itself can typically be expressed in a very small program and the presentation part can be chosen so as to minimise staff training costs[17].

2.5.3 JDBC

Inspired by the success of Microsoft's ODBC, Sun Microsystems defined the Java Database Connectivity (JDBC).

While ODBC is a flat library, requiring the user to pass context arguments on every invocation, JDBC wraps these same functions into an object orientated interface, so that the context is available to the library in an implicit way.

JDBC is a small step towards object orientation, because it models the underlying relational tables in the programming language, rather than modeling the programming language features (*Java* has a much more expressive type system) in the database.

[17] Typical candidates for presentation front-ends are spreadsheets and HTML browsers – interfaces that the users are familiar with.

The JDBC tutorial [FCHH99] explicitly claims that this is a low-level API on which other, more expressive interfaces are to be built and that one effort is underway to build a mapping from the rich type system of *Java* onto a relational schema.

2.5.4 ODMG

The ODMG Standard (a more detailed discussion can be found in 3.2) defines the interfaces and semantics for an object oriented database management system. Since the ODMG Standard is quite concrete, a number of projects [SR99, Rad96] have attempted to use it as an interface layer to a variety of object database management systems.

The ODMG Standard provides a uniform interface to various DBMSs and therefore is very similar to the ODBC and JDBC efforts. It differs in its expressiveness, by providing a vastly richer type system.

Where ODBC is a procedural API that can be used with many programming languages, JDBC is tied to *Java*, but presents the data in a much more convenient form for this object-oriented environment. The ODMG standard surpasses both of these in its closeness to the programming language. It defines a core object model and defines mappings from this model into several programming languages.

2.5.5 *SGML and XML*

The Standard Generalised Mark-up Language (SGML) as first described in [GR91] (first edition from 1974) and later standardised in [Gol86] was designed with the desire to insert machine readable information into otherwise unstructured text files.

Ten years on, computer science has matured a bit and SGML is being revised into the more concise, but nearly as powerful Extensible Mark-up Language, XML, as standardised by the World Wide Web Consortium (W3C) in [BPSM99]. While still true to the same ideals, XML simplifies SGML in several areas (for an elaboration, see [Fly99], section FAQ-DTDCONV) to simplify parsing and aid correct usage.

In order to illustrate the potential of XML, consider the following examples: One of the things that makes algebra tremendously powerful is the fact that so many of its operators accept and produce inputs and outputs of the same type: numbers. A further example of this principle are Ted Codd's relational operators. They can be combined in myriads of ways so that they always consume relations and produce new ones. My final example is the processing model of the Unix shell and utilities in consuming text files and producing text files, with the 'line of text' as basic unit of processing.

As described above and in most popular guides to XML, one could view XML files as text with markers. This exposition will present a different view, the view taken by the W3C: XML tools operate on trees in the mathematical sense. Each XML document is the representation of a tree, linearised by an ordered depth-first traversal, where three items are printed for each node: the begin marker, its contents and a closing marker. The begin marker can contain attributes to the marker type. The XML file format specification [BPSM99] provides a formal framework for constructing XML files. It specifies a BNF description including character sets and a small set of keywords, as well as the associated semantics.

XML files are comparable to programming language files. They need to comply with the specification both in syntax and semantics in order to be well formed. It may be a semantic error to refer to tags that are undefined or otherwise of wrong type. Tags are user defined and XML is designed to be a framework, in which those tags are used. In the very least, these tags must be used in pairs and pairs may be nested, but may not overlap.

The XML standard makes provision for specifying a BNF grammar with which the present document is purported to comply. This grammar can be partially or completely internal or external to the document. If provided externally, the file is called a Document Type Definition (DTD). DTD files are XML files.

Given the above framework, no tag types have been defined yet. The later specifications of the Hyper Text Mark-up Language (HTML) [RHJ98] are instances of XML DTDs. They define tag types such has 'heading', 'list item', 'quote' for intensional marking, types like 'center', 'table' and 'colour' for presentation purposes and visual components like 'image' or 'horizontal rule'. Despite their popularity, these uses are only a very small fraction of the possibilities offered by XML.

While HTML has evolved over time, XML has been designed for machine readability. It is envisioned that one (or a few) DTD files exist for each major problem domain where programs need to exchange data. Examples of such efforts are: the Precision Graphics Mark-up Language, the Digital Property Rights Language (for specification of ownership status), the Platform for Privacy Preferences (for specification of trade-offs of personal data vs. "freebies" in advertising), and the Open Trading Protocol to perform purchases among parties with varying payment and delivery methods. There already are hundreds of such standards being created for many different problem domains.

With so many tag sets being defined simultaneously, it is to be expected that name collisions will occur. The W3C Namespace Specification [BHL99] solves this problem by separating the name into two components, the namespace identifier and the actual tag. Any node inside the XML tree can bind a namespace to a particular prefix and use it inside itself and/or its subordinate nodes. One namespace can declare the same name with different types and meaning depending on the element type that it occurs in.

XML Infrastructure Standards

The advantage of having one single system to represent all of this information lies in the ability to share infrastructure. The W3C has standardised the following services to operate on XML files and within the XML framework (see figure 2.15[18]).

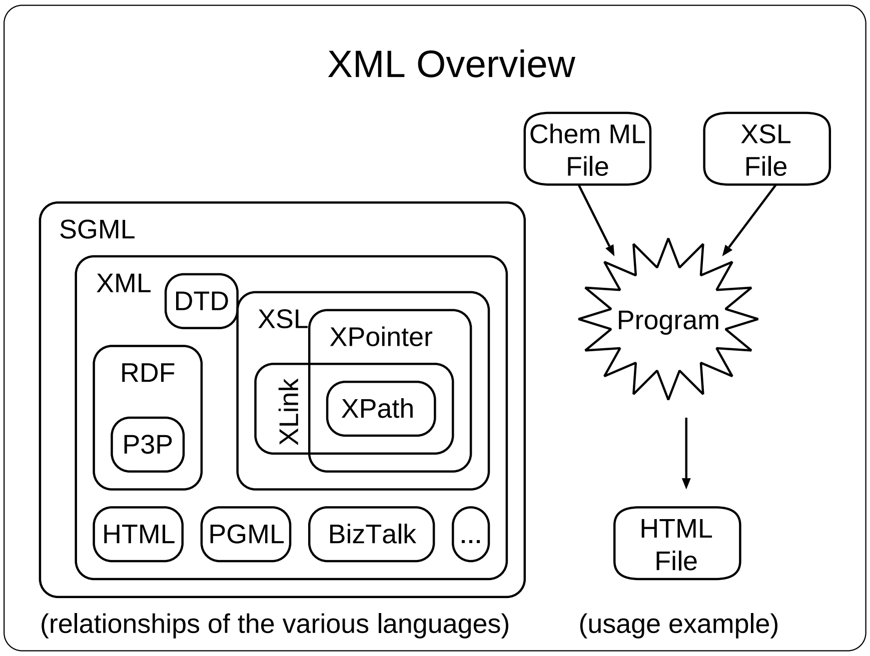

Fig. 2.15: XML Overview

The Extensible Stylesheet Language (XSL) [Dea99] is again a means of specifying presentation. The DTD for the XSL specifies a very general way of describing presentation objects, such as rectangular areas, area containers, tables, and so on[19]. The really interesting part of the specification with respect to general XML infrastructure is, however, a sub-document, describing XSL Transformations (XSLT) [Cla99].

The XSLT is a general tree transformation language, that operates in a functional fashion: typically a template is specified for the root node of the document and its transformations are carried out. If the command `<xsl:apply-templates/>` is encountered during processing, a template is sought for each child node in the input tree. The

[18] It must be noted that this diagram is not entirely accurate. While it is true that each of {XSLT, XPointer and XLink} can *contain* an XPath, an XPath on its own *is not* an element of the set of valid XML expressions. XPointer and XLink expressions are also not standards-conforming XML documents.

[19] One technology term frequently mentioned in the context of XSL is DOM, the Document Object Model. While XSL specifies elements for the composition of documents, DOM provides an API to manipulate the content, structure and style of these documents. It is mentioned here for completeness, not for its relevance to database connectivity.

results are inserted at the current processing position in the output tree. Possible operations include `<xsl:value-of>` for evaluation, conditional processing with `<xsl:if>`, repetitive processing with `<xsl:for-each>` and parameter passing to a child template (this is only an illustrative subset of XSLT).

XSLT needs a flexible way of addressing parts of trees. Since this need is shared with the XPointer and XLink specifications, the W3C have abstracted out the language to describe sections of the data. This standard is called XPath and can be found at [CD99]. An example path expression would be `child::para[position()=last()-1]` to select the previous-to-last of the 'para' child nodes. Several relative path expressions can be concatenated (the syntax dictates that '/' can be used as the separation character). The XPointer Specification [DeR99] allows addressing of arbitrary items, whether they be in XML form or not. This means a user selection or a page from a Portable Document Format (PDF) can be addressed with XPointer, while those things are outside the tree context in which the XPath specification operates. The last specification based on XPath is the XLink standard [DA99]. XLink allows resources to be associated with each other. It defines *simple* links that are uni-directional and embedded into the XML document that they are referring from, and it allows to define more flexible *extended* links that may be external and they may be multi-directional (in the sense that any participant in the link can be reached from every other participant). XLink entries can be labelled with a type, have *actuate behaviour* (either user activated or automatically activated), and have *show behaviour*, which determines whether the new content is shown in a new place, in place of the current element, or whether it is to be shown embedded in the current element.

XML Resource Description Framework

The XML Resource Description Framework (RDF) [LS99] specifies a way of recording meta information. An example from the W3C Metadata Activity Statement [Swi99] is:

```
<RDF xmlns:DC="http://purl.org/DC">
<Description about="http://www.w3.org/folio.html">
<DC:Title>The W3C Folio 1999</DC:Title>
<DC:Creator>W3C Communications Team</DC:Creator>
<DC:Date>1999-03-10</DC:Date>
<DC:Subject>Web development, World Wide Web
   Consortium, Interoperability of the Web</DC:Subject>
</Description>
</RDF>
```

Since RDF is built on the framework provided by XML, this is example is both an XML and an RDF file. It contains matching pairs of tags, most of which have attributes. The tags `RDF` and `Description` are part of the standard namespace 'rdf'.

The outermost entry in this example (RDF xmlns:DC=...) specifies that the tags specified in the file at the given location should be made available in the namespace 'DC'. This particular specification defines the tags `Title`, `Creator`, `Date`, `Subject` (and many others) for bibliographic abstraction.

The whole BNF for RDF contains 34 production rules, some of which refer to such things as 'any valid XML expression'. They define how to construct descriptions and classify the objects to be described into literals, IDs, references and collections of objects.

The RDF syntax allows for the specification of an *RDF Schema* which lays out the items to be described. The RDF file itself then contains metadata that actually describes some resource.

Note that the relationship between an RDF file and its RDF Schema is very similar to that of an XML file and its DTD. In both cases the files conform to a common BNF and in both cases is the descriptive one referred to by means of an include statement.

XML and Databases

Databases started out as high performance stores. Each store would be able to hold a very rigidly typed set of items where the type system was quite restrictive. Over time, databases have allowed increasingly greater freedom for the type system, most recently providing type systems as rich as that from object oriented programming languages, in some instances even storing the methods for the objects. This expressiveness is close to that of functional programming languages, where even functions are first class objects.

Databases still are most effective when they have very fine grained information about the structure of the data they hold. While it is possible to store raw text files, a database has greater scope for optimisation and providing expressive query facilities if the data is structured in smaller chunks.

XML makes the structure of conventional documents more explicit, allows for the enforcement of structure at the document creation level and thus can improve uniformity of the data provided in XML form. This is a way to bridge the gap between databases and unstructured documents from the document side.

3. OODBMS

3.1 *Several Approaches*

Object Oriented Database Management Systems (OODBMS) combine object oriented technologies (polymorphism, interface / implementation separation, inheritance, encapsulation, etc.) and database principles (persistence and consequently support for evolution, transactionally protected access to the data, system readable and interpretable type dictionary, as well as declarative query languages).

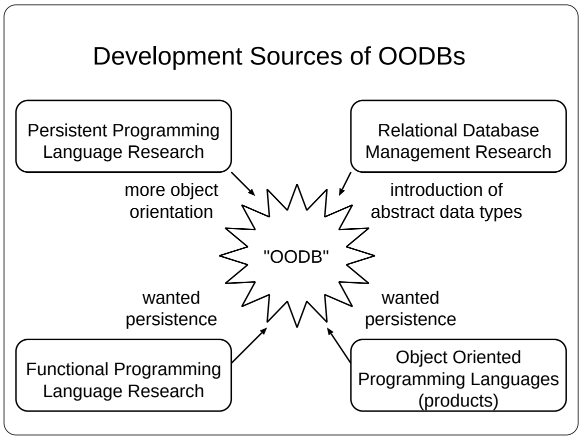

Fig. 3.1: Evolution towards OODBs

There is no single agreed meaning of the term OODBMS. Several groups have had needs for the features found in object oriented methodology while approaching the issue from their respective points of view.

The four most notable sources for ideas in the world of object databases are depicted in figure 3.1 and described in more detailed in the following subsections.

3.1.1 *OO Technology with need for persistence*

The first major attempt at a definition of what an object database might be, can be found in the "Object Oriented Database Manifesto" [Atk89]. The authors present 13 golden rules and a few optional features. The main requirements for a system to qualify as an object database by the standards of this manifesto are:

1) Complex Objects	7) Computational Completeness
2) Object Identity	8) Extensibility
3) Encapsulation	9) Persistence
4) Types and Classes	10) Secondary Storage Management
5) Class or Type Hierarchies	11) Concurrency
6) Overriding, Overloading, and Late Binding	12) Recovery
	13) Ad Hoc Query Facility

Rules 1 through 8 stem directly from programming language concerns. Rules 9 through 12 address database persistence issues and rule 13 is an attempt to describe a higher level interface than that of a programming language library.

The paper was published at a time when object orientation was mainly viewed in the light of features of the programming language and run-time system.

The manifesto recognised that a lot of programmer time was spent on building convertors to translate between the in-memory and in-database representations. Its proposals were aimed at having a database management system that provided equally expressive data types and computing paradigms as the programming language in use.

From today's perspective of object database features, the most interesting aspects of this proposal were (a) the fact that it proposed an object database in the first place and (b) that it required the ability to perform "ad hoc queries".

The implication of a single utility with sufficient generality to perform queries at run time is the following: the utility must be able to invoke user-supplied methods that were not available at the time that the utility itself was built. This ability is a major step over the the object oriented programming languages that were extended with persistence capabilities.

This capability requires machine-readable versions of the data-types. The database schema must record the name, number and signature of each method, including inheritance relationships. Only then can a meaningful request to a particular object be constructed. For an approach on how to do this, see section 7.5.1.

3.1.2 Relational Databases seeking more expressivity

Within a year of the publication of the position paper of the object oriented programming languages camp, the relational database camp felt the need to respond. The article "The Third-Generation Database Manifesto" [TaSRL+90] has been dubbed as "the empire strikes back".

The article, while completely foregoing the aim of the previous publication, namely to achieve a tight integration between modern programming languages and a database management system, the Third-Generation Database Manifesto focuses on selecting a balanced mix of the extensions to the relational model that have occurred since its inception.

While acknowledging the importance of encapsulation, this proposal does not go to the extent of supporting true user-defined object types. It suggests that the database management system be engineered in such a way as to be extensible (see figure 3.2). This way, the proposal argues, it will be possible to define new abstract data types, make the optimiser inside the system aware of the data type and bind it tightly into the system. This, however, does not quite coincide with objects in the sense of the word used by designers of object oriented programming languages. Object oriented programming languages typically provide objects with three types of property: behaviour (which in this case consists of the program module with which the database is extended), state (which in the relational world would be the contents of a record) and identity, which is the major dividing point between the two proposals.

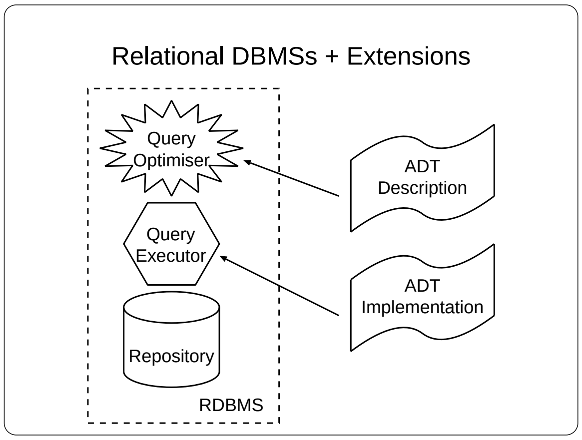

Fig. 3.2: Extensions for RDBMSs

The Third-Generation Manifesto does recognise the need for identities, but it does not make them a property of an object. It makes identities available as surrogate keys, which are items with a guarantee of uniqueness.

The Third-Generation Manifesto recommends many further improvements to the relational model, but these are not nearly as relevant to the discussion of object databases as the ideas of identity and encapsulation.

The difference in focus of the two proposals is clear and I expect that language integration will work better in an implementation of the Object Database Manifesto, because it matches the data models of the database and the language, while query processing, integrity checking and rule processing are likely to work better in this proposal. Query and rule processing benefit from optimisation, which is more easily possible in a compact, strictly controlled system with good statistics. This proposal is much more likely to build semantic descriptions of the object types for use of the optimiser than the Object Database Manifesto. Integrity checking will again be able to exploit domain knowledge better in the latter proposal, because better descriptions of the behaviour of the objects are given to the database.

3.1.3 Persistent Programming Languages using OO technology

One example of a well documented persistent programming language is PS-Algol [Atk88]. PS-Algol combines an imperative programming language with persistence. Persistence in PS-Algol is orthogonal to the data type (including user-defined types). It features the language concept of pointers, while disallowing pointer arithmetic. This means that each variable name is considered the root of a persistent tree and that a whole graph of items is made persistent by traversing everything that is reachable from the contents of this variable.

This notion of an *open database* provides storage and concurrency control, but does not provide declarative queries or access path optimisation. The major difference between this approach and the work presented in this dissertation lies in the fact that it does not provide ready access to the type of an object or any of its other metadata. While types, sizes and statistics could be computed, the infrastructure to do so is not available and the architecture is not optimised for this task.

In practice, PS-Algol maintains a mapping of the form (in-memory ID, on-disk ID) as shown in figure 3.3. Opening a database retrieves the root values into the named variables and all further disk access is performed on access faults from the memory management hardware.

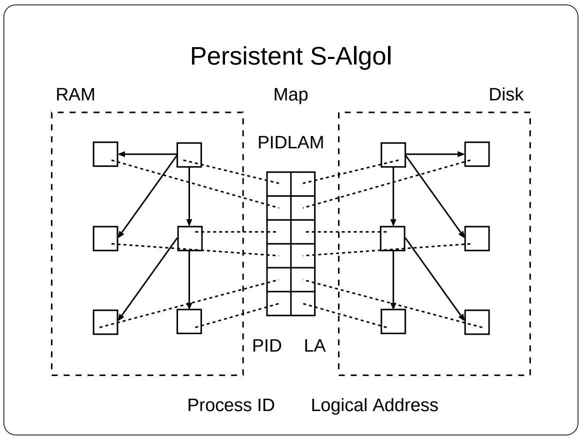

Fig. 3.3: The PID/LA Map

When PS-Algol writes records to disk, it replaces all in-memory identifiers by their corresponding on-disk counterparts. It also writes a type code that identifies the structure that is written. The read-time checking of type codes is not performed by the reading routines, but by the language interpreter, when it attempts to access the data in memory through a structure selector in the programming language.

The typical database guarantees of ACID are supported: atomicity uses shadow paging, consistency is achieved by faith (it is assumed that the program does 'The Right Thing'), isolation is achieved through locking of the whole database and durability is presumably achieved by the normal means of logging (although this is not detailed in [Atk88]).

PS-Algol is an extension of S-Algol. While the semantics are almost unchanged, it has a few additional library functions available to open and close the database, effectively manipulating the run-time system. Any S-Algol program should run in the PS-Algol environment unchanged.

PS-Algol has made persistent programming very much easier, but it is limited in its concurrency control and implementations of the language are limited by the orthogonal persistence, which places a performance penalty on all operations.

As an old, imperative language, PS-Algol does not provide facilities for modern software engineering: polymorphism, inheritance and separation of interfaces from implementations are not present. While the language itself has not been extended with these facilities, a language built on similar premises has been extended with the infrastructure originally developed for PS-Algol: Persistent *Java* [PAD+97, AJ99] is a modern programming language that also provides much of the functionality of pointers while disallowing pointer arithmetic. This is the precondition for the application of the PS-Algol persistence strategy. Persistent *Java* is a prototype programming system that again modifies the language run-time environment based on the principle of distinguishing persistent from transient pointers and maintaining a translation table. PJava separates class definitions from interfaces, allows polymorphism based on inheritance, and provides concurrent programming at the language level.

Neither of the two above systems maintain a database schema. Metadata is realised by the language type system, the types are stored in-place, object placement is performed on the basis of co-use. This is in contrast with previous DBMSs which would typically group objects by type and store the type only once. By not aggregating a schema, the collection of statistics becomes harder and exploiting this descriptive information for optimisation is thus infeasible.

3.1.4 Functional Database Systems requiring more structure

Functional Database systems are based on the Functional Data Model (FDM)[1]. The first complete specification of the FDM was presented by Shipman in 1981 [Shi81]. Shipman had produced a specification, but no implementation for the database language DAPLEX, which was designed to achieve good usability. It attempted to offer intuitive semantics for database specification.

As shown in figure 3.4, the FDM introduces *entities*, *functions*, and *constraints*.

FDM functions may have one or more parameters and are multi-valued. So-called 'single valued functions' are declared with a special syntax (a single headed arrow instead of a double headed arrow) and return a singleton set. Associated with the resulting set comes an entity type, which is the *Role* for each of the elements[2]. The task of the *Role* is to provide the entity type for the current context.

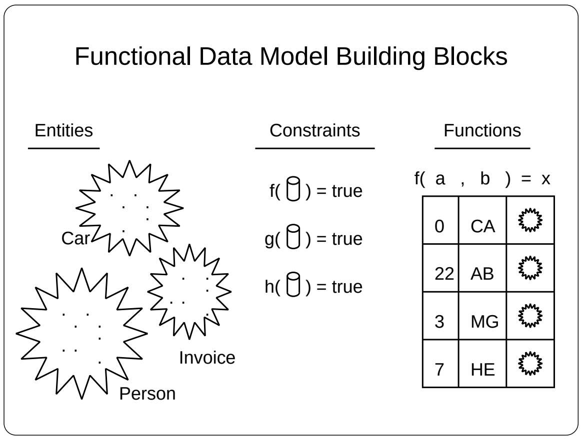

Fig. 3.4: FDM Building Blocks

The type system of the FDM operates by specialisation: everything is an entity of the root entity type `ENTITY`. From this root type, several system types are derived, so as to yield `STRING`s, `INTEGER`s, and `BOOL`s (amongst others), all of which are `ENTITY`s themselves.

[1] The purpose of a *Conceptual Data Model* is to define the domains and constraints for capturing the possible states of a section of the real world. It is important to note that a conceptual data model does not accurately model the real world, but that it restricts itself to the small part of the real world that is needed for the application(s) in question. A conceptual data model also attempts to retain independence from the physical data layout in a computer system.

[2] The `DECLARE` operator in DAPLEX is used to declare both, functions and entity types. If the `DECLARE` operator is used to declare a zero-parameter function, the system creates a new entity type instead

DAPLEX provides an explicit primitive (NEW) to create a new instance of one particular entity type. The approach for implementing this command is natural when applied to user-defined entities, such as EMPLOYEE, where it would examine the set of past instances (there is no way to destroy an entity instance once it has been created) and instantiate a previously non-existent one. The approach becomes artificial when applied to system provided entity types (such as INTEGER). As mentioned above, each set has an entity type associated with it that describes the *Role* of set elements. DAPLEX then provides a casting operator (AS), which selects all elements from this set conforming to a specific (usually different) entity type.

While Shipman's paper states that

`A single entity may be a member of several types.`

(p.146, section 2.5), there is no primitive in DAPLEX to associate a further type with a particular entity instance, nor is there a way for creating an acyclic type graph with confluence. As a result, in DAPLEX an entity instance can only have several types through sub-typing in the type hierarchy.

In FDM functions can be grouped into two classes: enumerated and derived.

Enumerated. functions are initialised to evaluate to the empty set for all arguments and in DAPLEX can be updated with the LET, INCLUDE and EXCLUDE commands[3].

Derived. functions are defined as an expression. [Shi81] describes, how DAPLEX requires an expression to denote the basis of allowed input values. It is not possible to unambiguously decide in all cases what an update to a derived function should effect. Hence DAPLEX requires an update procedure to be attached to each derived function before an update can be carried out successfully.

With the entities and functions available, a database can be constructed. The semantic expressiveness of this database can be increased dramatically by providing a set of *Constraints* that must be satisfied by the database at all times. A *Constraint* is any valid expression that evaluates to a singleton set containing either TRUE or FALSE. A database transaction would check all constraints at the point where all its commands have been executed and before the changes have been committed. If any of the constraints evaluates to FALSE, the transaction aborts.

While the FDM was very integrated and already contained some features that object oriented systems would require later on (entity instances could be compared to object identities), further research eventually tried to integrate its principles with object orientation. The most obvious omissions from the point of object orientation were:

Object Tying. Object identity finds its equivalent in the FDM entity instance. The state of an object is similar to the information contained in the various single

[3] While the LET command asserts the mapping from the given argument to the given set in the function, the other two commands modify the existing result set for the given argument.

argument functions operating on an entity. The behaviour of the object can be modeled through constraints. But the whole thrust of the object model, which ties these three properties closely together and allows their re-use through inheritance, is lost in the FDM.

Dynamic Binding. While DAPLEX allows the overloading of function names, each expression can be statically analysed to produce the function to invoke for this name.

Polymorphism. The polymorphism present in FDM is restricted to the kind provided by single inheritance. An entity can assume the `Role` of any of its super-types. A function declares the `Role` that it expects for each of its arguments. The set of entities passed into that argument are either cast into that role or omitted.

Interface/Implementation Separation. Since constraints can either be interpreted as providing an implementation or an (advanced) interface, it is not possible to have both of these facilities available simultaneously. It can be argued, however, that derived functions can provide any interface that is desired and that the signature of the definition for a derived function in fact is an interface against which further code can be developed.

One system, that is based on the functional data model and provides an extended DAPLEX, is the object oriented database management system P/FDM [Emb95].

P/FDM solves a number of the problems mentioned above:

- While entities are implemented as surrogates, as suggested by Shipman's paper, P/FDM also allows the association of keys with them. This allows for all the properties listed above, as well as making it possible to address a particular object.
- P/FDM introduces the specialised type of a *value entity*. Items of this type do not have a surrogate, since they can always be identified uniquely through their key (their literal value).
- P/FDM also supports the notion of overlapping subtypes, which means that an entity instance may belong to more than one most specific type. The problem of selecting a particular entity instance to have it assume a further most specific type is solved through the introduction of keys as mentioned above.

The functional data model does not support any other operations on the schema than the declaration of new types. An extension to the functional data model was proposed in the form of the extended FDM, or EFDM [KA86]. EFDM introduces special system-maintained functions that provide information about the schema contents. EFDM also provides constraints on the schema updates, and it allows deletion of schema data.

P/FDM still does not provide the view of objects as abstract types which can be accessed through a set of methods only, it does not provide dynamic binding (the type must be specified in an explicit cast), hence the DAPLEX polymorphism criticism applies. Finally, the interface/implementation issue remains to be argued.

3.2 The ODMG *Standard*

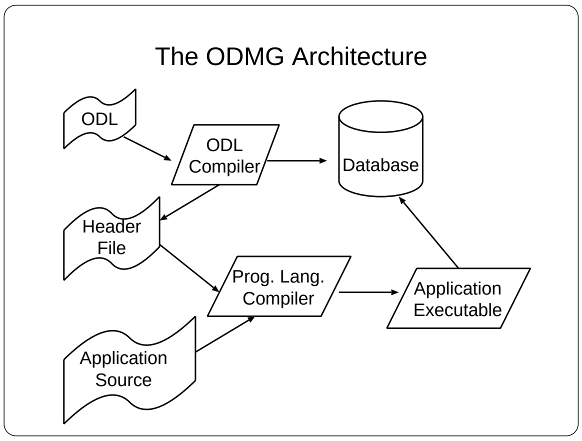

Fig. 3.5: ODMG Architecture

The Object Database Management Group was formed in 1992 by a consortium of object database vendors. The aim was to establish a standard for object databases. The reasoning was that such a standard would re-assure customers in their decision to commit to the use of such a product. Object databases were not widely available at the time and end users with problems that lent themselves to be addressed with an object database were reluctant to commit their fate to the product of a single vendor. With a standard, there is more hope that a customer could change from one product to another in case the supplier of the original product experienced difficulties.

Out of the four motivations listed above to build an object database, the ODMG Standard is based on the desire to first make object oriented programming languages persistent (originally, the languages C++ and Smalltalk were addressed, because those were the languages supported by the participants' implementations), to formalise concurrency control, aid program evolution, and finally achieve a greater degree of *data independence* through provision of an expressive schema and query language (the implementations preceding the standard had a minimum amount of schema support, but its usefulness was evident).

The latest edition of the standard is the *Object Database Standard* ODMG *2.0* [Cat97]. Former database standards defined three different languages (a Data Definition Language (DDL), a Data Manipulation Language (DML) and a Data Query Language (DQL)) with the associated semantics. However, these formally distinguished languages are not always presented separately. The ODMG 2.0 Standard defines ODL and OQL, the definition and query languages. The Object Definition Language will be described in the next section. The Object Query Language has been designed to fulfill several objectives: It aims to provide query and modification functionality, *SQL* `SELECT` compatibility and simultaneously retain simplicity. OQL is both a DQL and a DML. Figure 3.5 shows the principal components of an ODMG database system.

The ODMG standard does not incorporate any guidelines towards specifying access control.

3.2.1 The ODMG Object Model and ODL

There are three terms that need careful explanation before the advances and difficulties of the ODMG Object Model can be appreciated:

Type System. A *Type System* describes how to construct types. A type is a set of values. The set can be defined explicitly by enumeration, implicitly through storage limitation, or descriptively through an algorithm. Type systems typically provide *base types*, such as `int`, `string`, or `bool`. Type systems normally also provide *constructor types*, such as `struct`< ... >, or `set`< ... >. Higher order type systems include function definitions into the type system. Given those kinds of elements from the type system, it is possible to construct arbitrary types. Most type systems also allow that an individual, newly constructed type is given a user-selected name. It can then serve as a building block for further, more complex types.

Object Model. An *Object Model* is the description of architectural components present to structure the abstractions, data storage and computations in an object oriented programming language. It typically describes object existence by defining the processes of creation and destruction and by defining object identity. It describes the semantics of naming and access paths and it describes the available type system, as defined above.

Data Model. Finally, the term *Data Model* has been defined in footnote 1. While the type system relates to the theoretical well-foundedness of expressions and the object model describes the modes of computation, it is the task of the data model to describe a piece of reality in computer usable terms.

The ODMG Object model defines a type system that comprises base types, such as `string` or `object_reference`, constructor types, such as `set`< ... >, `struct`< ... > and `union`< ... >. Figure 3.6 shows how this object model relates to the data model and to reality. The construction of user types may combine these types repeatedly without restriction. The type system of course provides an object type `class`< ... >, which has some special properties: a class may contain an arbitrary number of entries, which may be attribute declarations, relationship declarations, or method signatures. Since an attribute may only be of the types constructed by the method above, objects themselves cannot be included as attributes. This also means that it is not possible to have a collection of objects, but only to construct collections of object references.

The ODMG Standard is multi-language aware. It provides bindings of its object model to different programming languages. Object class declarations contain attributes and method signatures, but the actual object behaviour is specified inside a particular programming language.

Lastly, the ODMG Object Model provides a single fundamental persistent object type, `d_Object`. Any object derived from this type is persistence capable.

Object creation, which primarily generates a locally[4] unique identifier, is of special importance: one of the advances of the relational data model was the introduction of *data independence* (see appendix A). While the relational model abandoned machine generated identities, the ODMG Database Standard re-introduces object identifiers and a facility to indicate relationships between objects. Object existence is controlled through reachability[5], where root objects are named explicitly[6].

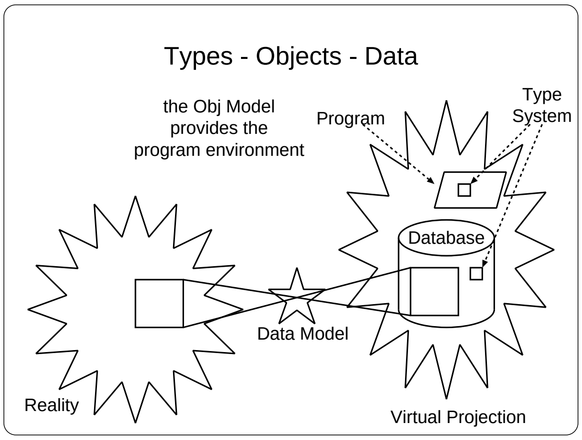

Fig. 3.6: Models and Systems

All type specifications inside the ODMG Standard are performed through Object Definition Language (ODL) specifications. ODL as a language can express a range of things, which the ODMG Standard classifies into *interfaces*, describing abstract behaviour, *literals*, describing abstract state and *classes*, describing both, abstract state and abstract behaviour. I disagree with this classification, because interfaces and classes are types, while a literal is a value and by talking about the abstract properties, we indicate an interest in the type only.

ODL allows for the construction of class hierarchies (with the persistent classes all rooted in `d_Object`) and interface hierarchies. The standards document specifies that classes can be derived using single inheritance and that interfaces may use multiple inheritance. Name clashes in the multiple inheritance hierarchy are detected at runtime during schema updates and prevented from ever reaching the database. The standard permits a class to implement several interfaces simultaneously, and also allows the omission of attribute and method declarations from the class when they are present in an interface that the class implements.

I suspect that this set of rules presents a compromise between the products that several vendors already had at the time the standard was written. I personally would much prefer a strategy where classes are used as a guide for the compiler on how to compose code, and where interfaces are used as contracts between different application subsystems. The resulting system would allow multiple inheritance (barring ambiguities dynamically at definition time) for interfaces, with those interfaces only containing method signatures. It would allow single inheritance for classes, where attributes and relationships would be inherited along with the method signatures of the parent class. Users of the objects would be presented with the interface only, while implementers would be presented with the class declaration. The advantage of this clear separation would be twofold: (a) access control could be implemented on a method basis, and (b)

[4] In the ODMG Model, Object Identifiers are unique inside a particular database.

[5] Please note the difference between *persistence* and *existence*. *Persistence* capability is conferred through inheritance from the persistence type `d_Object`.

[6] There are two types of root objects: (a) arbitrary user-named objects and (b) extents, which are maintained by the ODBMS and contain the Object Identifier of every object of the extent's type.

all modifications of an object would happen through traced channels[7], allowing active database functionality to be implemented.

One final observation on the standard is its specification of collection type constructors. In the standard, all collection types are themselves derived from `d_Object`. The declared aim is to allow a collection of items to be instantiated as an object. This inheritance relationship increases user convenience by removing the need to declare an object class. If this inheritance were absent, an object class with a single attribute would need to be defined. This attribute would be the collection. The collection would be addressed through the object identifier and attribute name, rather than invoking the collection methods on the object itself. This design choice has a severe impact on the remaining type system:

Since it is not allowed to embed objects within objects, it has now become impossible to create collections of collections. With this rule in effect, it is only possible to create a collection of object identifiers, which themselves represent collection objects. While this may be convenient for the implementation of the storage system and may force the use of less complex objects (which undoubtedly is good practice), it does break the orthogonality inside the type system. I personally prefer a more complicated storage system implementation to an exception in the type system. In practice, however, the current systems use the above specification and achieve very good performance with it.

3.2.2 The Object Query Language

In order to protect commercial investment in training of personnel and existing tools, OQL has been designed to accept any `SELECT` statement that conforms with the *SQL92* Standard [ISO92]. The representation for an *SQL* table in the ODMG model is a bag of structures. While accepting any valid *SQL92* `SELECT` statement, OQL extends the syntax, to allow for the following:

Query Definition. This facility allows parameterised queries to be defined with a name. Given that they are persistent, query definitions are very similar to *SQL* Views, but due to parameterisation they are more powerful. Their major advantage over views is the facility for recursive definitions. The standard neither explicitly demands, nor excludes the implementation of recursive definitions, but since they are such a powerful tool[8],

---

[7] Modifications to object state could either happen through a user-initiated method invocation, where code inserted at the interface level could trace the invocation, or they could happen through OQL statements, which again are under the control of the ODBMS.

[8] An example for the use of recursive functions would be the computation of a transitive closure: With a table `is_boss` listing the boss and the associated workers, the query `slaves(Joe)` would compute all direct and indirect workers of "Joe": `DEFINE QUERY direct(x) IS SELECT y.slave_name FROM is_boss AS y WHERE y.boss_name = x; DEFINE QUERY slaves(x) IS UNION( direct(x), FLATTEN( SELECT slaves(y) FROM direct(x) AS y ) );`

it would seem like a missed opportunity to omit them.

Dot and Arrow Operators. Attribute selection in classes or structures, relationship de-referencing and their arbitrary repeated composition in general path expressions are all performed through the use of the two equivalent operators `DOT` (.) or `ARROW` (− >). While *SQL* makes use of the `DOT` operator when a variable is used to denote a single tuple from a relation and a specific attribute of this tuple is to be extracted, OQL extends this use in a natural way to address attributes and de-reference relationships.

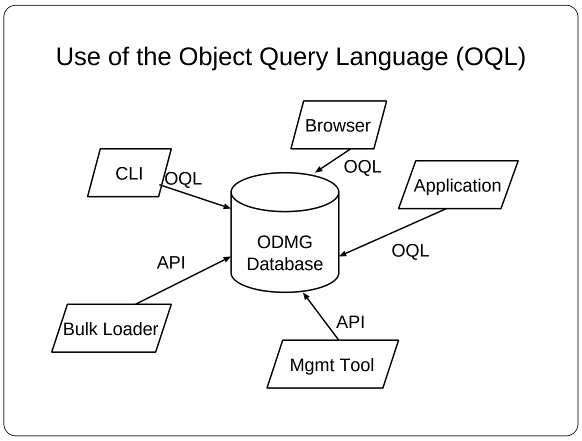

Fig. 3.7: Use of OQL

Object Creation. OQL provides a syntax for creating specific objects. If the statement to create an object is executed repeatedly (and the resulting object ID is retained, so that the object is *reachable*, then several distinct objects with the same state are created. Object existence is a very important topic and is addressed more fully in 3.2.1.

Method Invocation. The method of a particular object is again indicated through the use of a variable representing the object and one of the two equivalent operators `DOT` (.) or `ARROW` (− >) separating that variable name from the method name. Whenever a value is needed from an expression of this syntactic form, the appropriate method is invoked. Since several classes can define a method of the same name and signature the use of the same method name can lead to the invocation of different pieces of program code. Method names can thus be overloaded. The object model also supports the notion of several object classes implementing the same *interface*. *Interfaces* are not necessarily bound by a hierarchy (see 3.2.1). If a function argument is defined to conform to a particular interface, the most specific type (the object class) may differ for separate invocations. This allows ODL functions to be *polymorphic*.

Additional Operators. ODL introduces collection type templates and OQL adds operators to deal with them. Among these functions are `element` and `flatten`.

Modification operations, which in other databases are effected through the use of the `INSERT`, `UPDATE` or `DELETE` operations of the DML are now part of the query language, since method invocations allow for side effects to occur. `INSERT` operations for example, are performed through the invocation of the `insert_element` method on a collection object.

While the ODMG Standard calls OQL a functional language, I would rather refer to it as declarative: while nothing in the standard prevents functions (query definitions) from being recursive[9] or first class objects, this use appears not intended by the writers. OQL also allows (and indeed requires) side effects on method invocations to appear. Most functional languages have support for these features, which seems to disagree with the ODMG claim.

OQL is a practical compromise between backwards-compatibility and expressiveness. Its intended use is depcited in diagram 3.7. I would wish to see a future standards document to clearly state whether recursive functions are allowed. And if recursive functions are forbidden, I would recommend that a primitive be added to compute transitive closures[10].

3.2.3 The ODMG Metadata Interface

Databases conformant to the ODMG Standard store schema data, which is also called metadata, because it describes the bulk of the data present in the ODBMS.

The schema lists all derived types, exceptions, the class hierarchy[11] and the interface hierarchies. It is stored in the *ODL Schema Repository* and is made accessible through the same mechanisms as all other persistent data in the database. SQL has been very successful in providing access to data and metadata through the same interface. The ODMG standard attempts to replicate this success by also allowing inspection of its metadata through the normal data-access interface. The structure of the metadata classes is shown in figure 3.8.

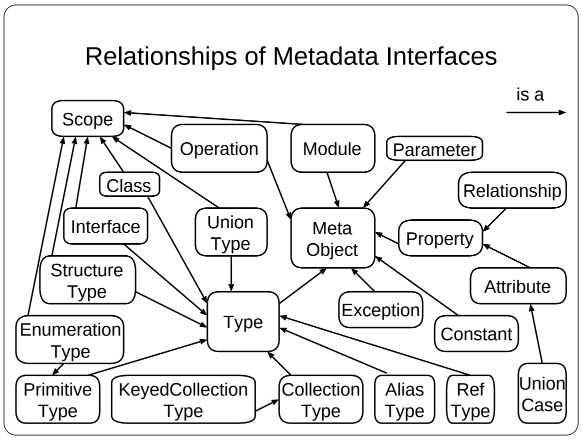

Fig. 3.8: The Metadata Interfaces

While the standard produces interfaces to access all the kinds of data communicable through ODL, it leaves the mechanism for obtaining something that implements these interfaces to the vendors. It would seem that there are again two ways of accessing this information: either through explicitly named objects, with the convention for the names specified by the provider of the ODBMS, or through extent names, which again must be specified by the provider.

[9] The standard claims in section 4.2 that it is not computationally complete, which would suggest that recursive functions are not permitted. A definitive statement disallowing recursion is absent.

[10] It is of course true that method invocation to a programming language will always permit transitive closures to be computed, but the same is true for the rest of the query language and one common problem tackled by OODBMSs are design databases with parts explosions which would benefit from such an operator. Incorporating this facility into the database would make optimisation more feasible, since estimations for its cost exist [LN89].

[11] The database schema is only concerned with persistent classes. All persistent classes of an ODMG compliant database are derived from the class `d_Object`, hence there is only one hierarchy to consider.

A second observation on the ODMG Metadata interface is that it only provides an interface for reading, which means that creation, modification, and deletion of metadata is up to the database provider to specify. By explicitly omitting the specification of an update-interface, the ODMG standard leaves these problems to be solved elsewhere. This involves the subject of schema evolution, which is probably going to be solved in a similar way as in the relational case: only restricted update operations will be legal, with additions to the metadata being much easier to implement than changes or removal.

The ODMG Metadata interface is lacking extensibility. As mentioned above in section 3.2.1, there is the object model, which is implicit in an ODMG Database, there is the type system, which is a part of the object model, and there is the data model, which is the description of which part of reality is represented inside the database. The data model of other database systems allows for constraints to be specified, which the ODMG Standard does not, and they typically impose access control, which again, the ODMG Standard does not. In *SQL* databases, extensibility is achieved through use of the same keys in the system catalog and related tables.

One way to offer extensibility would be to provide hooks (modeled as ODMG relationships) in the official Metadata specification to attach this additional data to. Adding hooks to specify metadata would break the uniformity of specifying all metadata in the ODL files. A second idea would be to offer text fields for annotation inside ODL. This, however, would lead to the need for separate parsers: the parser for ODL and the parser for each of the possible comment types. In my opinion, the most uniform way of offering extensibility would be through the equivalent of the 'C' programming language `#pragma` command. This would allow language extensions to be part of the parse tree. Since the structure of the schema itself is accessible through the metadata interface, querying the additional data would be natural.

4. EVENT DIRECTORY AS AN INSTANCE OF AN OODB

Chapter 2.2 has presented different types of event system, which were classified according to the following criteria:

- dynamism of communications paths (from very flexible WWW based systems to rigidly defined financial messages with access control)
- rigidity of message layout (from free text notes to *CORBA* type system conformant events)
- decoupling of participants (from completely decoupled game events to session-like database queues)

While the following considerations will to some degree be applicable to all of the event systems listed in chapter 2.2, they have been conceived in light of the local CEA system (see 2.2.3), which is based on the following assumptions:

- messages are strongly typed
- clients are 'disciplined'[1]
- access control (for reading, writing and service discovery) is an issue (as opposed to some systems which are aimed at information dissemination only)
- administration is grouped into domains

CEA encompasses an argument transmission scheme and several services.

Messages are instantiated from types in a user-defined type hierarchy. The transmission is channel-based, guarantees tunable detection of channel failures, configurably aggregates notifications, provides priorities at the message, client, or type level, is optionally source filtered, and incorporates role-based access control. Each of the services is either a producer, consumer, or proxy.

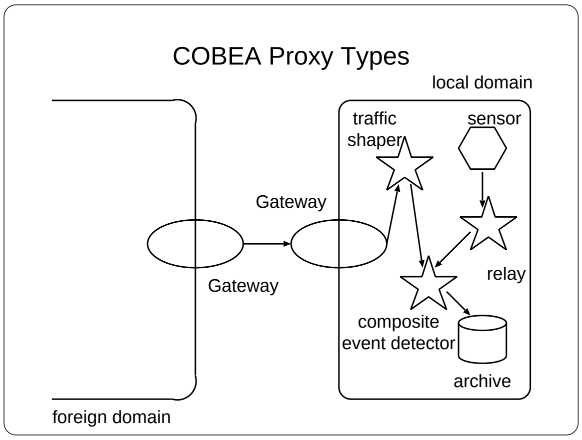

Fig. 4.1: CEA Proxy Types

The infrastructure provides information pipes with endpoints. This means that any processing element inside this path is a proxy. Some of the possible proxy types are shown in figure 4.1 and listed below:

[1] The way in which the Internet is currently operated means, that many people use dial-up connections on personal computers, where a 'session' could easily be terminated by hanging up the telephone or switching off the computer. The architecture investigated here allocates session resources in the server and does not model clients as frequently disappearing. Some remedies include time-outs and retaining the state in a storage hierarchy, but on the issue of frequently disappearing clients it still cannot compete with state-less servers.

- simple relay proxy for scalability
- composition proxy to compute aggregations from several sources
- translation proxy (gateway) to convert between domains
- persistent store, providing a time delay or replay facility
- filtering and traffic shaping proxy for manageability

The proxy types have been conceived over a long period of time, creating the expectation that the above list will grow further.

While it is possible to create a plan for event flows, produce a systematic naming and then set up a system from this plan, much greater use can be obtained from the system if it is more dynamic. For the purpose of dynamic creation, removal and connection of services, machine readable configuration information must be accessible. From the point of view of a user (programmer) of this information, a unified interface, preferably centralised would facilitate the use of all this infrastructure.

The following sections will present the considerations for storing typing information, how to build broker functionality, and how to describe the special class of producers/consumers that provide gateway functionality.

4.1 *Type Dictionary as* ODMG *Schema*

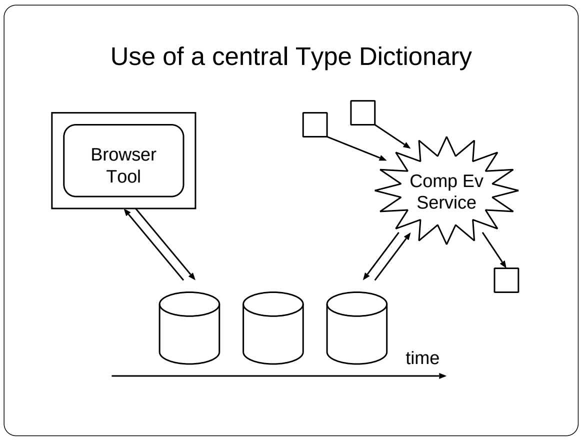

Fig. 4.2: Use of Type Dictionary

The first thing to consider modeling as metadata is the typing information for events. The type of an event primarily describes its internal structure. Programmers writing an event source or sink are provided with header files that define the concrete structure of the particular event type in use. However, having this information only in the form of header files is wasteful, because a much higher level description existed, from which the header files were generated: the definition files (typically *CORBA* IDL files). If this higher-level information is stored in a repository with a centralised interface, several benefits arise:

- Browsing tools (as shown in figure 4.2) can aid re-use by making it possible for a programmer to understand the structure of the particular type hierarchy for events, as well as making the specific details manageable.

- Composite event detection services can accept expressions with incomplete typing information[2], obtaining the complete type from a schema repository.
- Schema evolution is a hard topic and is currently only addressed through constraints on the operations available to modify a schema. If all event types in the domain are declared for inclusion in a machine-managed schema, then the conflicts can be avoided at declaration time. Updates to the schema are protected by a transaction and thus happen completely or not at all. Only when consistency is preserved will the changes find their way into the repository. While it may be possible to add a new sub-class, the constraints may prevent, for example, the declaration of a duplicate attribute.

Since CEA events are *CORBA* objects, the most immediate choice for a schema repository would be the *CORBA* Interface Repository. However, the approach chosen here is to store the hierarchy of event types in an ODMG Database schema. This choice is justified through a number of reasons:

- In principle, the ODMG Schema should be able to store everything that the *CORBA* interface repository stores, and more. Since development on the ODMG standard started later than development on the *CORBA* standard, and because *CORBA* gets more attention, the current specification of the ODMG schema is not as powerful as the *CORBA* schema. Since a lot of database query optimisation is based on accurate statistics and other metadata, I expect that the ODMG schema will eventually find a need for all the properties stored in the *CORBA* interface repository, and more.
- Our database implementation will go beyond the current ODMG standard, and provide active database functionality and access control through OASIS (see glossary), which itself is based on event technology.
- The query facilities for accessing the schema of a database are naturally more general than those of purpose-built repositories.
- With the event types known to the database, making these events persistent and querying them is possible without further effort.
- The effort needed to make CEA talk to a *CORBA* repository is the same as making it talk to an ODMG Metadata Schema.

Given those reasons, the following sections will address specific considerations of this strategy.

[2] If a particular event class A features attributes x, y, z, then an event expression $B(name) \Leftarrow struct(A.x, A.z) : name = A.x$ would not reveal the structure of A, but it is likely that the evaluator needs it.

4.1.1 An ODMG Schema as a Database Instance

The access to an ODMG Schema is defined through a set of interfaces. These interfaces allow access to the class and interface hierarchies, exceptions, auxiliary types, extents, and user-defined names. The natural way to represent the data behind the interfaces is through classes resembling the same characteristics and storing those classes in the database that they describe.

This approach of storing the metadata in the same repository as the actual user data has successfully been applied in relational databases that are based on *SQL*. The advantages of this approach are:

- uniform access control
- an interactive query facility
- uniform API for both data and metadata
- availability of a powerful query language
- transactional protection with isolation and constraints on schema updates

One argument that has been made against storing metadata and data in the same way was the difference in access patterns. It is expected that metadata will be accessed very frequently and thus it might be advantageous to lock it into memory. But since object databases will support much more complex data structures and consequently data items of greatly varying sizes, it would make more sense to allow an application to specify hints to the caching manager. The query optimiser, which is going to be one of the major applications using the schema, will then be able to get similar performance out of the schema as if it were locked into memory by design.

4.1.2 Present and Future Properties of Events

The metadata repository will primarily contain the information on which event types have been defined (type names), what the structure of each of these types is (attributes) and how those types relate to each other (inheritance). Besides this fundamental information, several other properties would be useful for the configuration of applications:

Default Priority. As mentioned above, priorities can be assigned per type, per client, or even on a per-event basis. The model clearly allows for a lot of customisation, but in the interest of uniform design, it would be desirable to specify all expected uses in the repository. The priority stored here would be the 'per type' priority. In order to be useful, this facility requires a global scale for the priority values. If relative ratings are chosen, events from different servers will not be comparable. An example of such priorities might be a high priority for a failure event and a low priority for a status event in a communications network reporting system.

Default Buffering Policy. Event sources may choose to retain a certain subset of the generated events. This could be useful for back-registration events (e.g. where a new client joins its connection to a game server and wishes to learn immediately about the state of the game, but the state is only available as deltas in the form of events). There may be some event types for which this buffer size is best specified in terms of memory consumption (a WWW application would be a good example, because any amount of buffer will be consumed, but the designer can at least provide a sensible guide). For other types, the default buffer is better described in terms of time (here, work-flow or email might be good examples, as they may be bounded by legal constraints).

Software Engineering Information. This may include the author, human readable documentation, files from CASE tools and anything else that pertains specifically to this event type.

Semantic Information. It might even be possible to keep semantic information such as an ontology, or at least constraints on the uses of a particular event class.

'static'	'dynamic'
Default Priority	Current Participants
Default Buffering Policy	Current Connections
Software Eng. Info	Usage Statistics
Semantic Info	Default Archive Location

The information described above is fairly static and exactly what one might expect a schema to contain. However, since a general purpose OODBMS is being used as the repository, it may be sensible to store configuration information, and make it available through the same mechanism:

Current Participants. It is beneficial to build a more fine-grained semantic model of the participants in the local domain. Simply recording that a particular process is a consumer and that another process is a supplier of events is useful, but can be improved upon. A management tool would be able to provide much better trouble shooting if the actual types of the applications (rather than their view towards CEA, namely as a source or a sink) were known to it.

Current Connections[3]. Again, from the point of view held by management tools, information on the currently active connections and their heartbeat and typing parameters provides a very rich basis for trouble-shooting. The number of events signalled is very much larger than the number of connections established, this allows the bandwidth use to be kept to a minimum.

Usage Statistics. Database management systems record statistics on the composition of the data that they manage in order to make good optimisation decisions. CEA would benefit in the same manner by deciding on how to best configure proxies, detectors and where to place filters. Statistics are supposed to give guidance only. They do not need to be up-to-date.

Default Archive Location. It is very conceivable that many applications will benefit from being able to utilise replay facilities. Each event type could be augmented with the specification of a default archive, along with a parameter to give a time or storage amount to indicate how many events to retain.

The properties listed above cite mainly a management application as their main users. However, the aim in storing them goes beyond this: next generation of applications will depend on the availability of this information in machine-readable form. Independent software agents will only be able to fully utilise the information on the network, if detailed descriptions are readily available. If the agent has to perform discovery all by itself, the programmer will have spent far too much energy on building something that should have been a generic service in the first place.

4.1.3 Retrieving Event Types

Retrieval of the typing information happens through the uniform interface provided by the ODMG Metadata Specification. The further data listed above should be accessed through the same interface, but the Metadata Specification does not specify how to, for example, provide statistics. If the ODMG were to provide, say, a `relationship`, to link each of the Metadata objects to a separate extension or description object, or provide some other mechanism to attach arbitrary information to a type, then the information listed above should be made available through those interfaces.

Until such a specification is available, implementation-specific *ad-hoc* interfaces must be used. In the context of metadata storage for event services, this information will be accessed through a higher-level API anyway, so it might be less troublesome to rely on these extensions.

4.2 Broker Functionality

The first step towards automation of the management of complex systems is to describe the services and the needs of the clients. It then becomes possible to provide some degree of automated configuration.

In the physical world, this metadata is stored at two levels of detail: in a yellow-pages directory and in a white-pages directory. In the CEA architecture, the same distinction is made. The white-pages service is a reliable source of concise and up-to-date information for a small number of clients. The yellow-pages service is a more far-reaching service, which does not claim freshness of its information, but rather tags it with the acquisition timestamp and an expected lifetime. While a white-pages service allows the

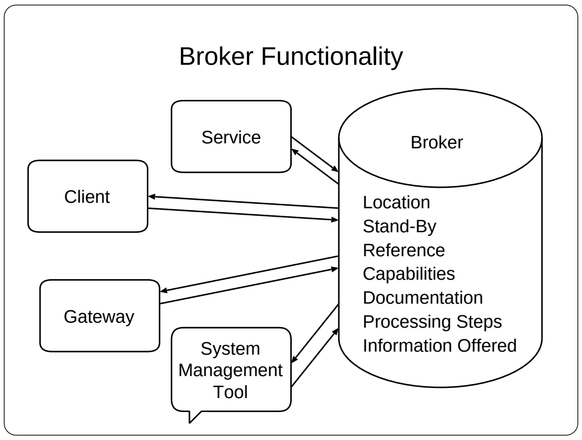

Fig. 4.3: Overview of Broker Functionality

matching of attributes in its index, a yellow-pages service employs information retrieval techniques and gives a much less concise answer, but rather gives a hint as to where to go and look.

4.2.1 Storing References to Participants

Since CEA specifies an interface in the *CORBA* IDL and a participant providing or using this service can then be specified by the means of a *CORBA* Object Reference, the first step towards broker functionality is to record the object references and provide a means to associate them with additional information.

4.2.2 Describing Sources

While the interface structure of a *CORBA* object reference is automatically available from the *CORBA* runtime system, this information is very low-level and general. In order to make meaningful selection decisions, it is best to have information available that is more matched to the task at hand. Figure 4.3 shows a broker with its data and its clients. In the case of CEA interfaces this information would initially be:

- Is it a source[4] or a sink[5]?
- What are the event types offered or consumed?
- What is the recommended heartbeat interval for each of these streams?

It is useful to store information about sinks as well as sources because then system management tools can more easily be built and operated. Knowing the consumption

[4] A 'source' is a supplier of events.
[5] A 'sink' is a consumer of events.

patterns and locations within the system might help to balance the load and decrease latencies.

A good example is a financial institution. In such a firm, event sources would offer notification of such diverse occurrences as dividend payments, bond maturation, market report availability, taxation changes and various deadline warnings. With the current push by firms like Microsoft, Novell and Sun to employ 'knowledge management', it might be very useful to mine the usage data for the quoted event streams. Doing so would reveal employees with competence or at least interest in certain areas. Making this information available in a directory could improve communications and efficiency in such a company.

Location

Location is an attribute often recorded in brokers. For CEA applications, it may be interesting to represent locations for different purposes:

Physical Location. The physical location of a particular sensor or output device is useful for management purposes. It can be represented at differing granularities, in different naming schemes (street / house / office / desk vs. world coordinates).

Managerial Location. This would be a description of the administration, ownership and jurisdiction to which the service submits.

Technical Location. Such a location description would indicate the networking connectivity of the participant.

Once the location of an object is known, its distance to other objects can be determined. Greater distances almost always imply greater cost. A trader will most likely have several possibilities for solving a particular request. Location can be used in determining an optimum for latency, cost and reliability.

Classification and Capabilities

It might be possible to create a classification graph for all the types of participants known in the current domain. If each registered participant is associated with a classification, then applications designed to work with the classification graph can make use of this information.

A much more general approach would be the use of an ontology which actually contains a machine-readable form of such capability information. Such an ontology would not only describe object classes with their attributes, methods and relationships, it would beyond that describe relationships among the types and constraints on the state that any object instance can assume. It might also describe how instances from the model map onto real world items. Typically an ontology is a classification hierarchy with unique labels for each node. Constraints, such as the fact that a particular type of car has an engine size in a certain range, or that a particular crop has a certain yield per acre, are used to provide machine readable semantics on the items in this hierarchy.

Currently there are several efforts under way to produce machine-generated classifications for use on the WWW. The classification graph described here could be generated by mining the schema information that defines the types that are used by the participants. Building an index from the names, comments and the structure of those types, information retrieval techniques could be used to cluster similar programs.

Processing Steps

For well-known services that actually form a part of the event system architecture, such as proxies, filters and composite event detectors, it may be beneficial to explicitly replicate some of the state of each instance, together with all available statistics. The reason again is to provide a uniform and centralised interface to increase the use of this information by making it more easily accessible. State that could usefully be stored in a trader would be a composite event expression, the list of current connections, and buffer sizes. Since only metadata is reflected to the broker, the amount of bandwidth required is kept to a minimum. Since CEA is based on *CORBA*, the standard transaction service can be used to perform updates in isolation. This prevents race conditions.

Additional Information

Finally, it would be very useful to have the documentation about each participant available in the repository. Beyond the documentation, other generic metadata, such as a preferred stand-by service could be represented within the repository, again with aim to have a machine-readable form of the information, so that any trivial task can be automated.

4.3 Describing Gateways / Domains

The connection of domains under separate management is most easily accomplished by the following method:

- The required resources are translated into a form understood by the local system, and
- the references to those translating resource interfaces are mapped into a convenient local namespace.

- Besides translation, it is possible to ensure some forms of quality of service through traffic shaping and map security control information such as authentication and authorisations between the two domains.
- The gateway will also have to feature a control interface that needs to allow to install new mappings and remove old ones. For this purpose it must talk to the type and configuration repository of the foreign domain.
- Given this design, it is possible to use gateways transitively, provided the gateway registers all the translation proxies that it provides with the local trader.

If such a representation is chosen, then the foreign domain suddenly forms part of the local domain and can only be distinguished from truly local resources by means of the namespace in which the item references occur.

Given that the gateway needs to register all its information with the repository anyway, it seems a good idea to use the repository as the persistence mechanism for the configuration of the gateway itself. In this way, the broad and powerful searching facilities of the object database, on which the repository is based, are leveraged again, by applying to everything that is provided and processed by the gateway.

5. WIDE-AREA EVENT SERVICES

Event based systems are particularly well suited for wide-area communications. The physical distance itself imposes communications delays, but event systems are asynchronous by nature and thus less affected by this problem.

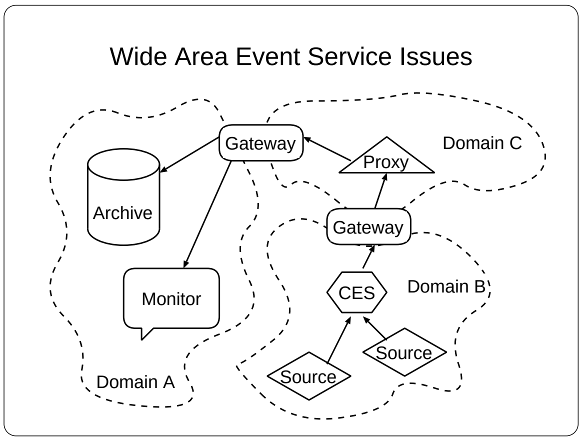

Fig. 5.1: Wide Area Issues

Figure 5.1 shows an example topology of interworking event systems and the following chapter describes some architectural components that simplify usage of an event system over a wide geographical area. Not surprisingly, they partition the administrative responsibilities, ensure uniform system views, enforce contracts among the participants and support heterogeneity.

5.1 Domains of Administration

Most open event architectures[1] will be administered along with the other services running on the same hardware. This implies that the event architecture should honour the administrative entities provided in the environment in which it is to operate.

Administration in such an environment will encompass the following:

- declaration of message types
- creation/registration and removal/destruction of service instances
- specification of access control and resource usage policy[2]

In order to cope with the complexity of managing all of those rights, rules, entities, types, etc., it is usual to abstract groups of services into service classes, to group users into roles and to group administrative powers into domains. This grouping is an abstraction and abstraction is one of the most successful tools that Computer Science has to offer in dealing with complex systems.

[1] Closed event systems will be used in a network specifically used for sensors etc..

[2] Since CEA allows prioritisation at the type, client and message level, it must be possible to specify the roles to the security system that may request those resource-intensive service levels.

Different systems offer this administrative abstraction in differing ways. Novell *Netware* allows for the construction of an administrative hierarchy, which delegates responsibility to sub-domains. *Windows NT* allows peering arrangements to be made, thus effectively creating a directed graph.

While CEA intends to provide the flexibility of an arbitrary trust graph, the problem of access control remains: the original source of information will have some disclosure policy. If a proxy, composite event service, gateway, or other type of relay is among the trusted recipients, then this relay will have to implement the same policy. This is relatively straightforward inside a domain: the principals are all named and their access status is defined for the original source, so if the access rules are published (which some sources may be reluctant to do), then trusted relays of any sort may use the same information as the original source to determine access permissions.

Outside a domain it gets more difficult. There are principally two ways of establishing domain interworking: either names are segmented and the software has been built to understand the semantics of names that name both a domain and a principal, or gateways map foreign principals and participants into a local, flat namespace.

While a simplified implementation is a very strong argument for the flat naming scheme, into which foreign names are mapped, I still believe that a two-tiered naming scheme which shows the domain name and the local name is far superior:

- It leads to the implementation of relative and absolute naming inside the role description language, thus increasing expressivity and re-usability.
- With a physical connection present, it becomes possible to address foreign resources without first building a mapping inside a gateway.
- It give one hint about the cost of an operation, since a local resource will on average be cheaper than a remote resource.

Further attributes (such as configuration, location, or capacity) would be available from a trader, which leads on to the next topic.

5.2 Finding Services

It has been mentioned above that machine-readable descriptions of all components are going to further automate and increase reliability. From the perspective of an application or application designer it is going to be most convenient if all that information is accessible from a single source.

At the same time it is a given in any system that is dispersed over a wide area that components are going to have independent failure modes. Figure 5.2 shows an example topology and how information can be replicated. From that perspective, it would be useful to distribute any vital resource, so that not all of it becomes unavailable.

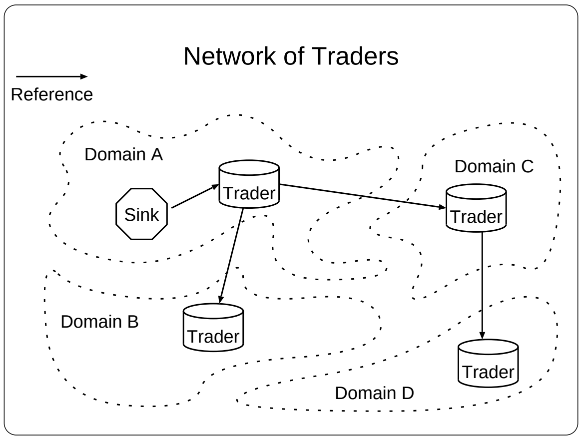

Fig. 5.2: Traders Providing System Image

The architecture promoted here will achieve both those aims in the following way:

- Each CEA participant will export a method to obtain a handle to 'the' metadata repository.
- The reference returned will provide an ODMG interface to the database of properties (names, locations, rules, configurations, etc.).
- This reference may be generated dynamically, it may be remote, or it may not refer to an actual repository containing all the data. But none of these restrictions are going to be visible to the user.
- Since the mechanism for obtaining this reference is the same for all participating entities, it serves the same purpose as a centralised, well known repository. The difference lies in the fact that it will continue to work, even if a partitioning of the network occurs and it is cut off from the rest of the world.
- Relative names may still be very useful, especially if a convention is layered on-top of this system. In such a way, names like 'help-desk', 'postmaster', or 'MOTD' (see glossary) may be defined and functional independent of location.

This system is inspired by the domain name service, and can draw upon experience with it: the DNS attempts to resolve requests locally, caches answers to requests that had to be handed on, and uses time-tagged data to manage cache expiry.

The difference between the DNS and the architecture presented here lies in their respective structure: the DNS is organised hierarchically, while this network of traders consists of an arbitrary directed graph. This graph is not a connectivity graph, but rather a graph recording the current trust relationships.

The tasks of the trader can be summarised as follows:

- Provide the interface to some local metadata repository (types, configurations, entities, rules, etc.).
- Resolve logical names, such as `local_domain`.
- Provide a facility to get the handle for the trader of a particular domain by domain name or other attributes[3].
- Provide searching over everything[4] in reach. The information may be cached with timeouts, but will mostly be generated "on the fly". This system faces the same problem as any loosely coupled database, namely, that the output it produces is derived from data outside its own control. As a result, *ACID* consistency can reasonably be guaranteed inside a single query, but if that query is run inside a transaction, it may not be possible to ensure that all participants in the wide area honour the transaction protocol, thus requiring a loosening of the consistency guarantee or facing an increased transaction abortion rate.
- Now that 'domain controllers' can be located and the wide area searched, the trader can build on top of this and provide the facility of constructing a 'translation path' terminating in a local participant. This means searching the trust graph for a viable route to the desired object and initiating the construction of translation services to convey the identity, role, and data type for the connection[5].
- The trader in charge of building a 'translation path' will make some attempt at global optimisation, eventually producing some execution plan. In order to reify this pipe, the individual connections and possibly components will need to be constructed. Allowing remote traders to request such resource use would require complex policy management at each service. It is likely that this policy management would then be separated out of the services and instantiated as a free-standing service. Having an autonomous policy management system would introduce a new translation problem. The ideal location for performing policy management on the creation of new services is the domain trader, since it will now handle both access control to the actual data streams and access control to the resources necessary for translation. This feature can then be used by local and remote services. The tension between traders being autonomous and wishing to inter-work is balanced by having tunable access control, that can accurately describe, how much information is to be divulged upon outside requests.

[3] The two extremes to the spectrum of solutions are an omniscient service, like *AltaVista* or the *WHOIS Database* at the one end and the introduction of impurities into pure domain names to give an indication of location on the other end.

[4] The structure of the database will impose restrictions on this, so that domains have to be addressed explicitly. It would not be possible to ask for 'all types, anywhere', but for 'all types in domain X'.

[5] These on-demand constructed services may wish to include a self-destruct facility which de-registers and de-allocates them as soon as the heartbeat feature indicates failure.

- With these facilities available, further automation may be possible, in that a client may specify that it is looking for a particular service and the service gets created on demand, policy permitting.

The above trader relies on the availability of the appropriate metadata for each of the tasks at hand. The more data is stored in machine readable form, the more of the above become possible.

5.3 Mismatch between differing Event Systems

One of the premises of event systems is that interconnection is more easily and reliably possible than in a more powerful object system. This section investigates the problems that can occur between two sites. At first, it is assumed that the sites are under independent management, but employ the same underlying event architecture, while the second part considers divergence in technologies as well.

5.3.1 Distributed Mismatch in homogeneous Architectures

The four main problems during schema combination are as follows (see also [Col97]):

Matching Attribute Names. Attributes may carry the same name in two domains, but differ (`name` may refer to a type of tool in one domain and a person in another domain)

Differing Attribute Names. Conversely, two attributes may carry different names, but refer to the same entity (`name` in one domain may be called `surname` in another).

Matching Attribute Types. While being labelled with the same type (say `string`), the mathematical domains (used set of values) may differ (the string `os10000` in one domain may be expressed as the string `oseidel` in another domain).

Differing Attribute Types. An attribute type may be chosen differently in different domains (`ZIP` may be a numeric value in one domain and a string in another). The problem of mapping the identifier for an entity in one domain to that of another (`os10000` vs. `oseidel`) is subsumed by this case, where the real types differ (one collection could be called strings-a and the other collection could be called strings-b and these could be special enumeration types).

The arbitration of those problems can be performed in a number of ways, with the two extremes being the bottom-up method, and the top-down method. In the bottom-up method, an event class hierarchy is matched against another one by building explicit conversion functions for each pair of corresponding classes. In the top-down method, structure and meaning are forced by convention that everybody has to adhere to.

This is a trade-off between programmer work (and thus practically correctness) and flexibility.

5.3.2 Impedance Mismatch between Architectures

As mentioned in section 2.2 asynchronous messaging comes in various flavours (MOM, tuple spaces, event systems). Each of these systems has advantages not supported by the rest. Figure 5.3 shows a variety of transitions between different event systems and which transmission property is lost.

While some features, such as typing information, can be encapsulated in a message for 'tunneling', others, such as transactional transmission, are part of the infrastructure and cannot be included.

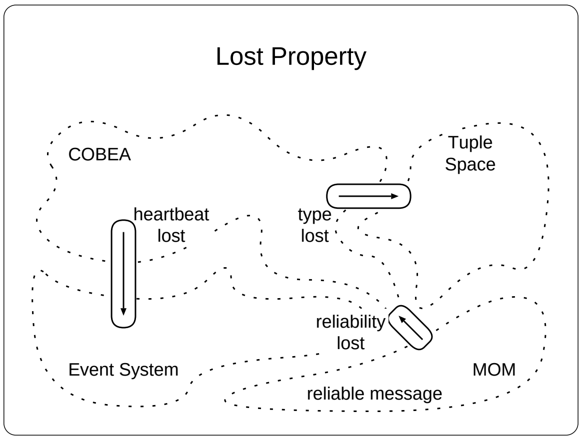

Fig. 5.3: Mismatch of Properties between Messaging Systems

One thing a system can do under the circumstances of feature loss is to document it. This leads to the question what this information should be attached to. Clearly, a trader listing participating components would also note their properties, thus allowing the necessary information to be extracted. But while such a documentation of behaviour may help a trader build a path, this may reduce usage of this information by applications. The idea of attaching this information to the stream end-point is less useful in the presence of combinator services: if a combinator relays events from several streams with differing properties, it is not clear which property is to be attached to the output channel.

The chosen approach provides a field inside a message that documents the transmission properties that effected its delivery. The previously mentioned combinator services not only have to compute a composite timestamp, but a set of composite delivery properties to attach to the output event.

The loss of some particular properties may be alleviated as follows:

Reliability. CEA does not guarantee reliable delivery. 'Reliable' in the strongest sense is associated with *exactly-once semantics.* While CEA makes no guarantees, it is fairly simple to construct Forward Error Correction (FEC), by sending the same data repeatedly and it is also possible to enforce *at-most-once semantics* by building two proxies that perform numbering and serial verification.

Typing. Systems that omit typing information may easily be instructed to carry extended data in which the typing is encoded. It may also be possible to provide the typing implicitly by opening a single channel per type.

Uptime. In the extreme case, the CEA heartbeat protocol may be interpreted as *per-link flow control.* If the interruption notification of CEA is really desired, then a gateway can be constructed that will provide the same information across other systems.

In summary, applications can only react to this sort of information if it is made available and the ever-growing cadre of 'adaptive agents' typically relies on an environment that provides all of this information.

5.4 Gateways

Gateway services in the CEA architecture are participants. Unlike the trading services mentioned above (which serve metadata), gateways serve data. They are purely conceptual and their work is performed by standard components available in the architecture. Their advantage lies in the added description. The components making up a gateway could be combined and configured in a multitude of ways. Labelling this group of components as a gateway restricts the number of possible configurations and thereby makes this 'macro' more predictable.

A gateway contains two components: a traffic shaper, which is used to limit the resource usage of the external participants and a composite event service, which is used to translate from one event type to another, possibly using external functions.

The fact that these two items are combined in a 'macro' and the fact that the uniform meaning of this macro across all domains restricts the possible contents of the CES (=Composite Event Services, see section 7.2) actually adds information.

This particular use of the CES has a single source and it may be possible to describe the translations in a separate language, specially tailored to this task. The advantage of a translation language would be that it is limited to a set vocabulary and functionality, which allows all external functions to be provided in the CES. If implemented in such a way, the gateway could be described by the translation expressions that it supports and a much greater semantic meaning about it could be exploited. If the lower level CES expressions were used, less meaning would be preserved and an automated system would need to incorporate much more logic to interpret the description.

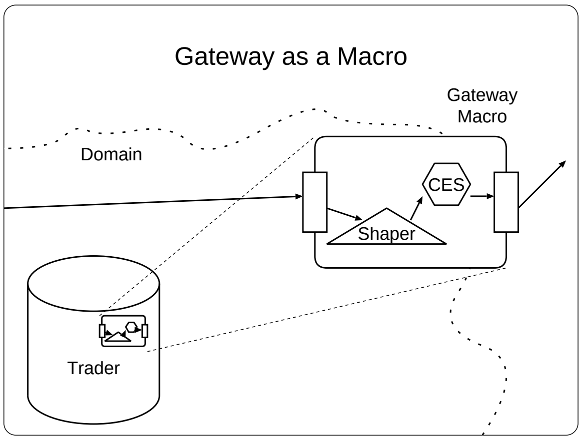

Fig. 5.4: Gateway Macro as an Abstraction

A different way to explain this advantage is to say that this macro presents an abstraction: two flexible and powerful components are used to implement restricted functionality and they are encapsulated in a module that presents a simplified interface. Figure 5.4 shows one such abstraction and how the extended properties of such a macro are recorded in a trader. The complexity is hidden and emphasis is placed on the functional interface.

As mentioned above, tools can be tailored to use these modules, but the benefit also extends to interactive management and visualisation applications.

6. DISTRIBUTION AND FEDERATION

Distribution is a broader concept than *Federation.* Distribution describes any partitioning of a task in space. Distributed tasks may be performed in parallel, but need not be. Problems can be partitioned along two dimensions:

Task. It may be possible to partition a large task into several sub-goals to be solved and distribute the work to solve these sub-goals across several solvers (a solver may be a machine, a process, thread, a remote interface, etc.).

Data. It may also be possible to have instances of the same service in several locations. In this way the partitioning is performed on the data.

Federation is a form of distribution that partitions the data into autonomous management domains (see [She91]). The interface provided by each of the participating solvers is the same, but the implementation, policy and functionality may differ. If the interfaces differ, then they are typically surrounded by a wrapper to provide a uniform interface.

While many distributed systems are planned using a top-down design model, federations often are formed by getting several independent parties to agree after they have started offering services. Figure 6.1 contrasts the difference between those two approaches. In those cases, the interfaces usually provide an option to express the 'unavailability' of particular functionality. Applications built upon those interfaces must make a different set of assumptions: namely, more care must be taken to deal with the case that a certain operation or property is absent from any node in the federation.

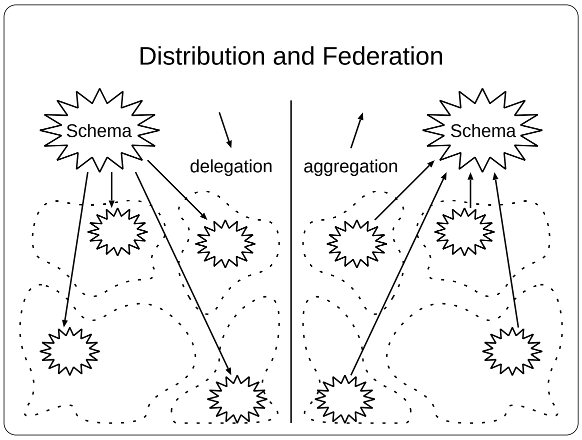

Fig. 6.1: Distribution & Federation

The architecture presented here is intended to augment an existing infrastructure (*CORBA*) and provide additional functionality (similar to [Hul97]). It aims to provide a seemingly closed environment in which all interfaces are unique. At the same time, it must acknowledge the existence of a multitude of co-existing systems. This problem is approached by providing a component architecture that allows external services to be encapsulated or augmented by components inside the local domain. Local applications can make use of these services without worry that an alternative interface may yield greater success. It is the task of the environment to ensure the presentation of services in a familiar way.

The problem of incomplete service remains. It may be possible to 'tunnel[1]' transactional transports through a domain that does not support transactions[2], and deal with other omissions in a similar way. Some omissions, however, cannot be rectified. In those cases, a typical solution is to provide the illusion of the existence of the functionality, but without implementing it. This leads to the malfunctioning of applications and affects robustness.

The solution presented here lies in the fact that individual services have been separated into components and may be treated independently. By providing convenience functions[3] that allow the application to test whether a certain property holds for the intended participants (e.g. 'Does the connection from A to B support exactly-once semantics?') and by separating the properties into individual services, it is possible to provide applications with graceful failure much more easily.

The key to achieving this illusion lies in the provision of machine readable descriptions of the system capabilities and the system configuration.

The system envisioned here is to be used in a large number of places, each with a separate schema. This will make it unmanageable for everybody to participate directly in a federation. The spectrum of costs for interaction ranges from $O(N^2)$ where a convertor is needed between every pair of participants, to $O(N)$ where a convertor is built between each participant and a central, unifying schema. Designing such a global schema for all conceivable application domains seems an impossible task. There are several intermediate solutions: the schema space can be viewed as a graph, and as long as a path exists between two vertices, a translation can be performed. The problem here is quality control: not every translation may be of good quality and measuring that quality in the first place will be difficult.

This proposal therefore presents the strategy of a forest topology:

Figure 6.2 shows how similar schemas are grouped together and abstractions are formed. Each abstraction is a federation schema. The example in the diagram provides two federation schemas ('money' and 'science'). The topmost abstraction for each topic is the root of that tree in the forest. Examples of trees might be a collection of schemas

[1] A transactional transmission requires the two endpoints to agree on the outcome. A transaction protocol between the two end-points can provide the required guarantees, even if the underlying transport is unreliable.

[2] An example scenario might be the offering of event services inside and outside of a firewall, where no alternative routes exist.

[3] A 'convenience function' is a service that could be performed by a client, because the client has access to all necessary information. But it might be wasteful and error-prone to re-implement such functionality repeatedly. An example for such a function in the above context might be the finding of a path through the type graph.

for factory control, another one for online shopping catalog updates, or possibly for security violation notifications.

The advantage of a forest structure over a general directed graph is that it enables a trade-off between the use of competing implementations and the advantages of being able to exploit common structure.

Individual schemas are confined to their private namespace. A 1:N mapping from name-spaces to Internet domain names exists, but the namespace is tagged with a generic name, namely the first Internet domain in which it was used. All names are normalised (in the sense that each namespace inside the name is renamed to its generic name), so that they become comparable.

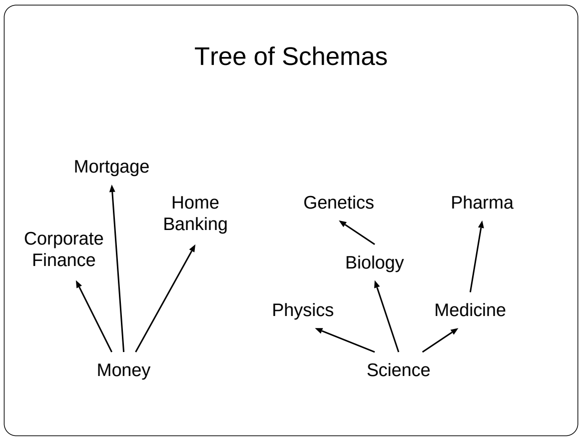

Fig. 6.2: Tree of Schemas

This description has now proposed a tree of schemas. Each schema itself is a tree with event classes as its nodes. The global tree describes a very general schema at its root and derives more complex schemas through addition. Schemas at more concrete levels of the tree include a complete copy of their more abstract superiors. While a programming language may use these inheritance properties in order to construct concrete code only at compile time, the system presented here consists of a replicated database, with a star-wise distribution policy: changes to the upper-level schema are distributed outwards to the more concrete schemas. Each lower schema contains a complete copy of its parent.

The concurrency control strategy of this method is well defined: the structural aspect of the architecture enforces that updates can be made in one place only. This one place is the one that has control of the schema. Schemas further down the tree are derived, but under the control of a separate entity. Hence, concurrency control on this data structure means that there is one well-defined writer and possibly several readers.

6.1 Schema Evolution

The principles for schema evolution in the context of this distributed event architecture are the same as those encountered in object databases. The major areas of focus are:

1. Modifications to the class or service type hierarchies:
   - addition of classes (including copying)
   - (with sufficient information) garbage collection of classes
   - re-parenting of an existing class
2. Creation of a new version of a particular class:
   - addition of methods / fields
   - modification of method signatures / field types
   - removal of methods / fields
3. Creation of a new version of a particular service type:
   - modification of composite event service expression
   - modification of default parameters

While general objects can contain arbitrary method code and their attributes can have very specialised meaning, events are more likely to be transparent. One example of this transparency is the composite event service, which relies on the fact that manipulation of attribute data from events is semantically valid.

With this advantage, namely that the architecture describes the meaning of events more elaborately in a machine readable form than a general object system does, some of the hard schema evolution problems are alleviated.

6.1.1 Naming

The implementation of the federated schemas operates as follows: each class and service type is assigned a unique identifier. This identifier is composed from two parts: the originating domain and a unique number within that domain.

The names at the programming language level will stay the same and re-compilation of existing programs should produce output involving the latest type version. Every handling of events, however, should check the event type identifier and reject wrongly typed input.

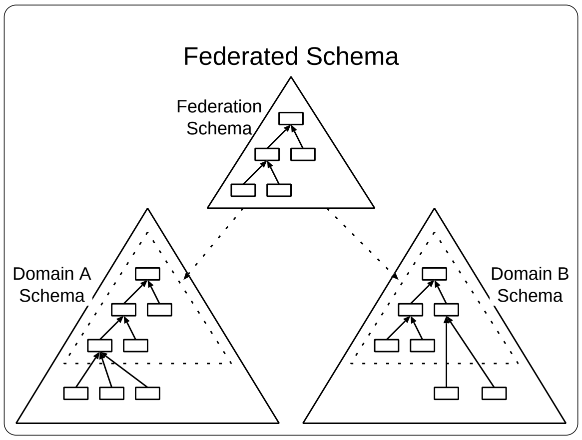

Fig. 6.3: Schema Federation

Every modification to an event class creates new versions of that class and its derivatives. Creating new versions means that the type identifiers of old versions stay the same and that the new classes receive new type identifiers.

Since the structure of the schema is such that part of the schema tree is under control of an entity higher up in the federation hierarchy (see figure 6.3), schema servers lower in the hierarchy register interest in modifications to their parent servers. The complete schema of a parent server is cached locally, but the identifiers are retained.

6.1.2 Naming Clashes

A naming clash occurs when a parent server adds a class, but a class with the same name has already been defined in a derived schema. Conceptually, the name is an attribute of the schema entry and the schema entry is uniquely identified. Practically, declaration of a class leads to the creation of programming language header files. These header files must include name-spaces (language-provided or pre-processor created) that indicate the domain of the schema. Normal scoping rules will then allow the local name to be provided as the default, but the higher name to be accessed if necessary.

Addition of a new attribute with a duplicate local name into a parent class causes greater difficulty: the intuitive solution would be to honour the authority of the federation. However, renaming the local attribute may possibly cause existing local applications to still compile, but malfunction. Given that convertors between types are generated automatically, it is much easier to insert a convertor that translates between events with differently named fields while preserving application correctness. A sensible way to deal with such a clash would thus be to rename the newly added parent attribute and retain the name of the child attribute. If it is possible to issue a schema update which simultaneously removes an attribute and also removes the renaming of a parent attribute, this automatically generated renaming scheme can be removed. Now applications (or more specifically an application's ODL usage) need to be tagged with the level of the schema hierarchy that they are designed against. If the schema in the current context performs any renaming of attributes, then the application should at least be stopped from compiling or at best those mappings should be masked automatically.

6.1.3 Addition / Modification / Removal of Classes

Given the active nature of the whole architecture, it is natural for the repository in control of a derived class to register interest in change notifications about the parent class. Whenever such a notification arrives (it may come from the local repository, or from a remote one), a new version of the derived class is created. But the system can do better than that: since event services form a part of the architecture, it may be possible to create two new service types in the repository immediately, one for each direction of conversion (from the old to the new version and reverse). The deletion of an attribute leads to a convertor service type with perfect output, while attribute addition will lead to a convertor service type that adds a default 'unknown' value (the reverse is true for conversion from the new to the old type). If all events contain a system attribute that describes the information quality[4], the convertor can be designed to indicate this fact.

It is enough for a convertor type to exist: a trader will then be able to traverse the graph composed of event classes as vertices and convertor types as edges and instantiate a convertor proxy once it requires such a translation to occur.

6.2 Access Control

Access control in any system involves a three-dimensional binary matrix that lists the principals, resources and operations, with the intersection specifying validity of a particular access.

A well-known example is presented by file-systems, where all possible combinations of the operations are enumerated and thereby compressed into a single integer. The remaining two-dimensional matrix, populated with integers, is then either sliced along the resources, to produce a list of principals with their access rights (this is then called the Access Control List (ACL)), or it is sliced along the principals, to produce lists of Capabilities that each principal has.

The advantage of ACLs is easy revocation of rights, while the advantage of capabilities is easy delegation.

[4] Examples for information quality indicators may be the authentication of a trusted source, default values, timeliness or known measurement errors.

The method used here is called OASIS (see [Hay96]) and operates as follows: a (potentially multi-valued) function is used to compress the large number of principals into a small number of roles. Similarly, a (potentially multi-valued) map (an enumerated function) is used to compress the large number of resource- / operation- combinations into a small number of access control policies. The remaining matrix of $roles \times policies$ is very manageable and is stored explicitly.

Practically, OASIS principals wishing to perform an operation request a capability from an OASIS server. The OASIS server performs tests on the presented credentials and registers interest with the issuing authorities of those credentials to be notified of any changes in their validity. The capability is then used to access the desired service. A session based service caches the capability and registers interest in changes of its validity, while a state-less service checks its validity upon each invocation.

This access control scheme combines the quick revocation feature of ACLs with the low overhead of checking capabilities. It also avoids the problem of principal naming, since that is completely up to the OASIS security architecture, which in turn leaves it up to independent authentication services.

The specification of OASIS policies may involve arbitrary computations, such as times, load of the system, login location, or any other attribute of relevance.

The operations in the federated event system range from schema access, over discovery of system participants, to access to event streams.

OASIS, as a federated system itself, makes it easily possible to set up explicit and flexible trust relationships between domains, and its event-based alarm system fits well with the remaining infrastructure.

6.3 Quality of Service

If the resources were under-utilised at all times, and complete information were available in all circumstances, QoS would not be an issue. Since this is not the case, QoS interfaces attempt to provide as much metadata as possible, in order to allow adaptive applications to produce the best possible output or presentation in the given environment.

The indicators available in the federated event system are as follows:

Data Quality Description.

Heartbeat Rate.

Delivery Priority.

Bandwidth Reservation Service.

Transmission Semantics. (best effort, at least once, at most once, exactly once)

Delivery Receipts.

These indicators may be attached to a particular event, a particular service, a link, or an end-to-end path. Depending on their host, they may or may not vary in time. The obvious advantage in the context of an event system is that notifications happen asynchronously in the first place: this is not a change in paradigm as it is in an operating system providing QoS feedback where the active entity is typically the program itself.

6.4 Infrastructure / Efficiency

The infrastructure for the system described in this chapter provides for peer-to-peer event services. Only the schema repositories form a hierarchy of federations. Schema repositories hold fairly static information, while traders hold configuration information. Applications interact with traders only. It is the trader that performs the work of discovering a source according to a declarative query and then constructs an efficient path of conversions from this source to the application that made the request. While traders hold up-to-date configuration information, they utilise centralised caches that operate much like Internet search engines. These search engines perform regular crawling of the traders, extracting all accessible information and indexing it for information retrieval purposes. Since these search engines will mostly be used by automated entities (the traders), it is again helpful to store as much metadata as possible (e.g. any configuration information that is cached must be tagged with a timestamp of its validity).

This scheme ensures local independence and quick response times within the local domain, while trading off wide reach across domain boundaries against freshness (because the state in search engines is not updated synchronously).

When a trader is requested to locate an event source, it may have several options. In such a case, optimisation is applied on several levels: if several conversions are performed, it may be possible to combine the conversion expressions into a single conversion, thereby gaining efficiency. This, however, will only be possible, if the security system allows the transformations. It may therefore be possible to pay special attention to the design of access control policy so as to allow for such optimisations. Another issue is quality, in that some paths may degrade the service.

7. THE EXPERIMENTAL FRAMEWORK

For the design of computer systems, as mentioned in section 3.1.4, it is often the case that a model[1] describing a section of reality is created. An analysis of what might take place within this model can then be carried out.

Practical examples of scenarios for event system use would be (ordered by the uniformity and control exerted, from the most to least controlled environment):

- inside a single application (e.g. a custom built process control system)
- inside a suite of applications (e.g. in an office suite)
- as part of a telecommunications network infrastructure
- as part of a home appliances network standard
- inside an industry standard (e.g. trading offers from auction services)
- as part of an "end user client" Internet standard (e.g. active content)

These scenarios raise interesting issues with respect to potential problems and the patterns of use to be expected: clients may frequently disappear without observing a protocol that would allow resources to be freed, the system may produce bursts of occurrences, and finally it may be hard to name mobile participants.

The system presented here is targeted at applications in the latter half of the above list. There are also lessons to be learned from some wide-area systems (namely, the Domain Name System, the World Wide Web and Electronic Mail) that are not event driven:

Scalability. While the WWW is a great success socially, it has been plagued by technical (scalability) problems. These problems are now addressed through proxies, load balancing, and replication.

Legibility. Both the WWW and email are successful in utilising both human readable names and protocol commands.

Transparency. End user applications for the WWW have added an indicator to show whether a transfer is still in progress. This suggests that advanced error reporting functionality (such as throughput rates, transmission segment loads, server utilisation) may be helpful if available.

---

[1] Such a model would contain entity names with their properties, relationships between these entities, constraints, and possibly references to further descriptions.

Independence. While the DNS 'only' allows delegation of responsibility, WWW and email applications allow for completely de-centralised control. The underlying network itself relies on peering agreements to achieve connectivity.

Given the above observations, the current chapter will investigate components of the CEA event architecture and show how they draw on this experience.

This chapter describes which components are available as part of the infrastructure and how they can be combined to provide:

- seamless interworking of separate domains (translation)
- access control with different levels of trust and authentication
- a unified application program interface
- generic value-adding services (event composition, traffic shaping, debugger, browser, archival, statistics)
- different service levels (persistence, prioritisation, latency/bandwidth tradeoff, confidential tunneling, heartbeat monitoring)

An example scenario can be seen in figure 7.1:

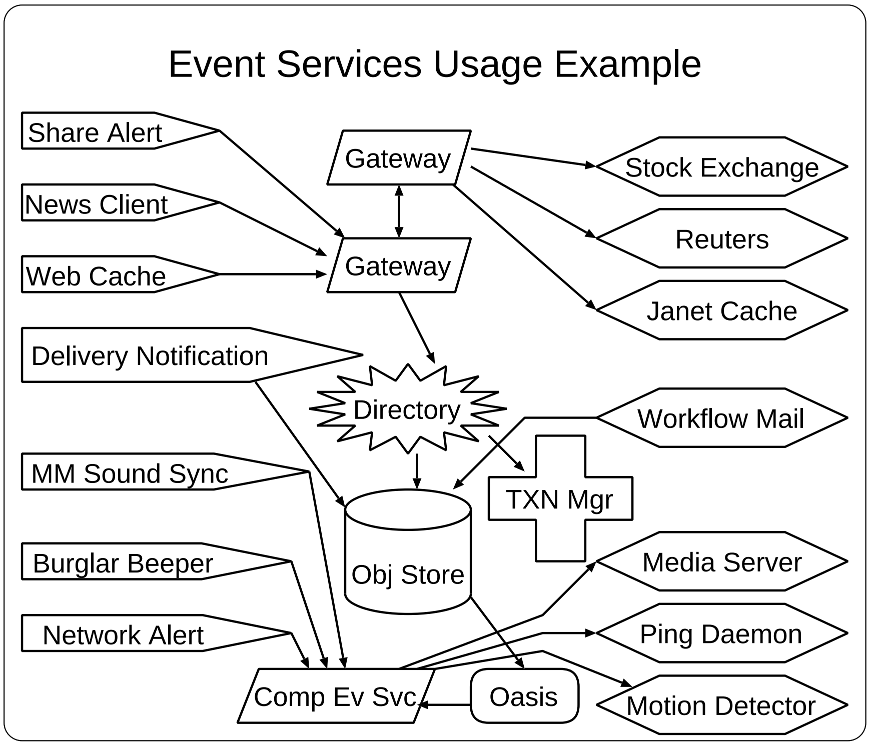

Fig. 7.1: Overview Example

In figure 7.1, three types of entity can be observed. Clients are the programs, which *register* interest in event occurrences. Servers *publish* the types of events that may be notified to (authorised) clients and the remaining Services form the general-purpose infrastructure that enables this technology to be used. The individual sections of this chapter will detail the design of each logical component; each may have been implemented from several self-contained building-blocks.

7.1 Event Services

As is often the case in Computer Science, this project makes use of layering as a tool to manage complexity. Initially, basic interfaces are defined to perform event handling, later on those APIs are utilised by higher-level services to provide a powerful infrastructure.

The model for distributed communication used in the approach presented uses asynchronous messages. Several clients can receive the same message and a single client can receive messages from several different servers.

The CEA architecture builds on the standard *CORBA* infrastructure. It has been developed by several people in the OPERA group.

Please note that there is a small but distinct difference between the *publish/register* approach presented here and the so-called *publish/subscribe* model. The word *subscribe* suggests that a client, once aware of the existence of an event stream, may unconditionally join the stream. The approach presented here, however, allows the publisher control over the set of its clients. Once released, the use of information cannot be controlled. This system incorporates a mechanism to generically specify which clients are to be trusted and in which way. Encryption and a Trusted Computing Base (TCB) at the client may increase the level of trust that is placed into the remote location.

7.1.1 Event Transmission

Events, which is a short form for *event instances*, are objects, whose state and identity is marshaled into a neutral[2] form and then transmitted using a streaming protocol.

7.1.2 Event Source

Upon startup, the source will publish an interface reference to the trader. In doing so, it will refer to *event types*, notification of which can be requested through the published

[2] The marshaling format is neutral with respect to programming language, operating system, and underlying hardware.

function. The interface type is known to the schema repository, where the trader can acces it. It has been added by the stub compiler.

Upon invocation of this function by a client, the source will perform the necessary authentication and create a virtual circuit conforming to the requested quality of service. Both participants have the event type information available before the actual transmission of event data.

As is the case with the *CORBA* Event Services, CEA event services can be composed from the outside. This requires more complex interfaces, but allows for automated construction of data flows.

7.1.3 Event Sink

Each event sink conforms to the published event-sink interface. It first acquires an object reference to the source object, along with the necessary event type description and access control capability. When it invokes the registration interface, it passes its own object reference (for the subsequent signalling), the type of the events and a filter expression for what it wants, and its access capability.

7.2 Composite Event Service (CES)

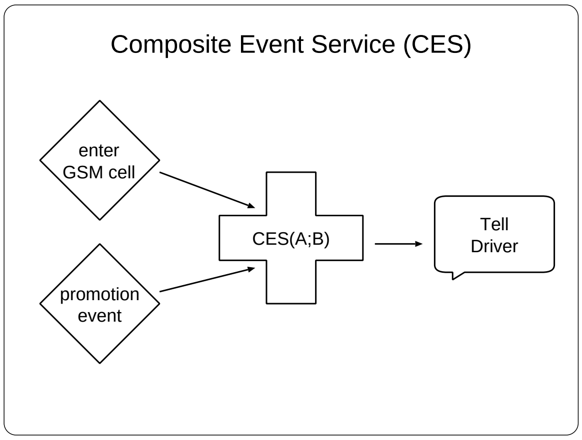

Fig. 7.2: Composite Event service

The composite event service is a generic event processor. Since the structure of event instances is available to the architecture, the composite event service can extract data from arbitrary events. Figure 7.2 shows an example scenario in which events from two separate sources are used as filters for each other, so that a combined notification will only reach the recipient if they match. The CES provides a special language to describe how streams of events can be filtered, re-combined, or translated. Several of the other services in the architecture are based on the CES, but the CES is available for use by end-users as well.

The composite event service is a process that gets passed one or more event source identifiers and appropriate filter expressions, as well as the necessary access capabilities. It registers interest in the events described by the filter expressions and starts receiving the events. Besides those sources, the CES also accepts a composite event expression, which is drawn from a simple language:

operator	meaning
$type(V_1, V_2, ...)$	basic expression to match an event of type
(A)	grouping expression X
$X;Y$	expression X occurs followed by expression Y
$X - Y$	expression X occurs before (without) expression Y
$X\|Y$	expression X occurs, or expression Y occurs
$X\{Z\}$	expression X occurs and condition Z is satisfied

An example composite event expression might be:

$$meeting(A, B) := seen(A, R); seen(B, R)$$

This example illustrates variable usage. While the variables A and B are declared as parameters on the left-hand-side (LHS), the declaration of variable R happens dynamically. During evaluation, both A and R are bound to values, when the first $seen()$ event is observed. The expression above is then evaluated in the same way as:

$$meeting(A, B) := (seen(A, R_1); seen(B, R_2))\,\{R_1 = R_2\}$$

The example above also illustrates another problem with composite event detection: the consumption policy. It is possible to perform a new evaluation for each event that arrives, it is possible to use each event several times, but only once per matching template ($seen()$ is a matching template), or it is possible to use each event at most once per composite event expression. Another question arises from the possibility of receiving a series of events of types:

```
A A A A A A B
```

and attempting to match them against the expression $A; B$. Should the first, or the last or some other A event from the above series be chosen in the expression? At the very least, these A events differ in their timestamps.

The composite event detector in use here employs the simplest matching strategy, consuming each event only once from a particular stream (two composite event expressions may each register for notification with the same filter from the same source, but will then be fed different streams). The first matching event from a sequence is chosen.

The timestamp could sensibly be computed as the maximum of the timestamps involved in the expression, and this is exactly what our composite event detector does. Other systems generate a new timestamp from the local clock at the CES.

7.3 Arbitration and Translation

The arbitration across domains is discussed in sections 5.2, 5.3 and 5.4. The fundamental problem lies in the inconsistency of naming and meaning across management domain boundaries. The two extremes for ensuring correct operation across boundaries are (1) a globally enforced ontology or (2) a complete set of convertors between any pair of corresponding event types. In many simple cases, an event *seen*() with parameters *person* and *room* has the same meaning in two domains. But the event system generates unique type identifiers which would still ensure that those two event types are distinct. A simple translation expression:

$$seen_1(P, R) := seen_2(P, R)$$

is enough to bridge the gap between domains and satisfy the strong type checking, but this expression must be available somewhere. It is expected that this mapping is performed by hand at least once.

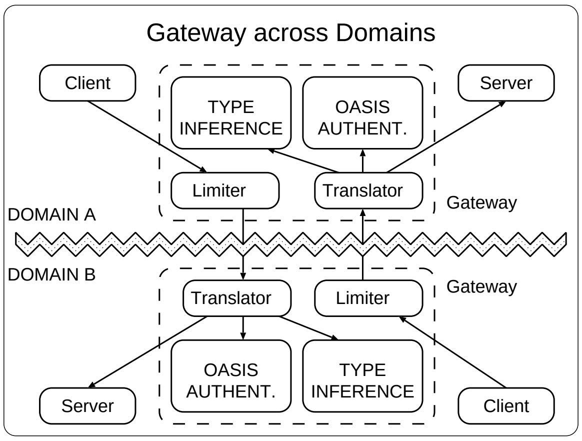

Fig. 7.3: Gateway Internals

Figure 7.3 depicts two gateways across a domain boundary. The flow of an event stream is achieved by first discovering the correct type of event to deliver (TYPE INFERENCE). This process is a search across the graph formed by the schema and the available translator expressions. Once the OASIS credentials have successfully been verified, event delivery can begin. The source itself provides a certain level of service, which is implicitly controlled by the credentials presented at registration time. This protects against resource abuse. Similarly must a receiving system protect itself against an overwhelming number of events delivered from a source system and thus employs a bandwidth limiter that algorithmically discards excessive events. These two separate functionalities have been grouped into a single service because they determine the amount of resource usage caused by outside influences. Having a single service in control of this functionality improves manageability.

7.4 Directory Services

There are three types of directory service, each managing data of increasing volatility. While actual event channels transmit user data, the services presented in this section hold metadata only. Types are long-lived and configuration information is short-lived. While the schema repository and the trader must contain up-to-date information, search engines trade the freshness of the information for coverage of a wider area. Thus, a second level of metadata is introduced: timestamps and expiry dates on the metadata make a statement about its validity.

7.4.1 Schema Repository

Chapter 4.1 describes how an ODMG database can be used to hold a schema repository for the CEA. A functional schema repository was developed during the research presented in this thesis. The schema repository is structured as in figure 7.4 and holds all information related to the schema:

- a local type counter
- federation-defined types
- user-defined types
- federation-defined classes
- user-defined classes
- inheritance relationships among the classes

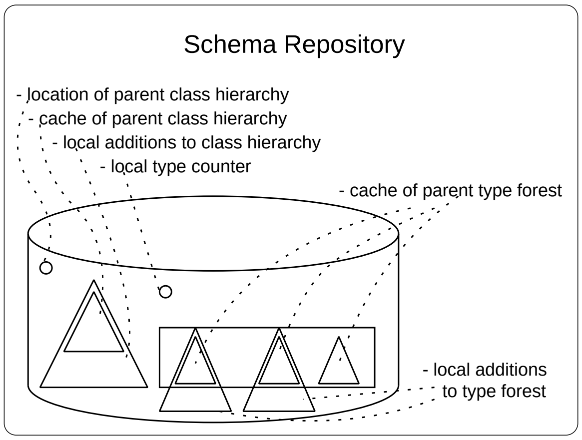

Fig. 7.4: Schema Repository

Since the repository is an application that uses an ODMG database, updates to the schema are protected by transactions. In particular, consistency is preserved, which means that several constraints ensure that the schema is a possible configuration.

The major task of the schema repository is the maintenance of federated consistency in the face of schema evolution. It has an autonomous type counter and produces new versions whenever a modification is applied to the schema. Modifications can be initiated at a higher level in the hierarchy or at the local instance. Those modifications are propagated to dependent schemas, but not to parent ones.

A further use for an object schema is to store the object behaviour, the method code. The ODMG database architecture omits the storage of method code, since the standard supports several languages and it is not possible to judge from which language the class will be used. It is then not possible to define the clear semantics of the code and thus it is better to leave it transparent to the programmer to choose and see which code will be executed and in which language environment.

7.4.2 Trader

The trader provides a white-pages service. It is structured as shown in figure 7.5 and is in charge of holding an up-to-date version of the local system configuration:

- a list of search engines
- a hierarchy of service types
- a list of instances for each service type, with configuration
- a list of channels attached to each instance

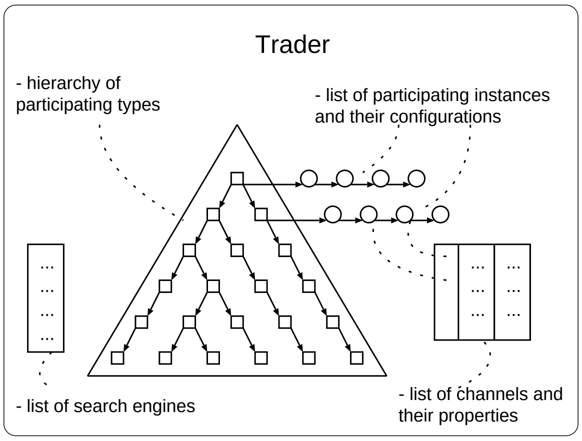

Fig. 7.5: Trader

A trader with this information offers a single point of reference to get a concise, machine-readable overview of the systems operating in the local domain. This is very useful for independent software agents, network management software and human systems administrators. It operates in the way described by section 5.2.

7.4.3 Search Engine

The search engines are based on their WWW counterparts: they collect data by crawling across traders. Figure 7.6 shows how they are integrated into this infrastructure. Since each trader refers to some neighbours, it is hoped that the graph of traders is not partitioned, thereby facilitating a complete coverage of the whole graph from anywhere. Search engines employ the same technology as traders, in that they match typing information and class attributes accurately, but they also employ information retrieval technology to access the descriptive fields. A search engine is a yellow-pages service.

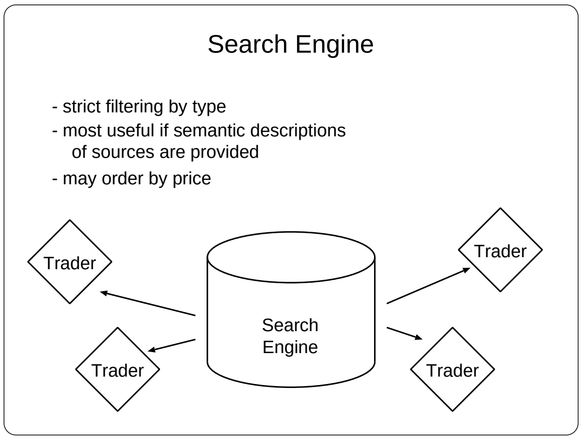

Fig. 7.6: Search Engine

The difference between a search engine and a trader as mentioned above is the following: a search engine covers a vastly greater area and would thus be a hot-spot and bottleneck, if update of the search engine were mandatory. A search engine collects data at a slow, and independent rate from other configurations in a domain. It keeps timestamps on everything and if available, it collects validity-timespans. This allows resources in remote domains to be discovered, but gives no guarantee of existence or accuracy. Just like WWW search engines, such a service is nonetheless useful, if employed properly.

7.5 Persistence

The persistence mechanism provided within the framework of the CEA is based on previous work of the research group [Lo94, Tho90]. A diagram of its operation can be seen in figure 7.7.

The system operates as follows: the Byte Segment Custode (BSC) maintains system-enumerated byte arrays. It guarantees atomicity of write operations. Together with the Log Manager, the BSC can also provide the durability property of transactions. Access control is performed using the OASIS mechanism and structure is imposed on the byte arrays using an extension of the Structured File Custode (SFC). Object state is marshaled for storage in this structured storage and it is possible for very large objects to have only updates migrated to disc, ensuring economical resource use.

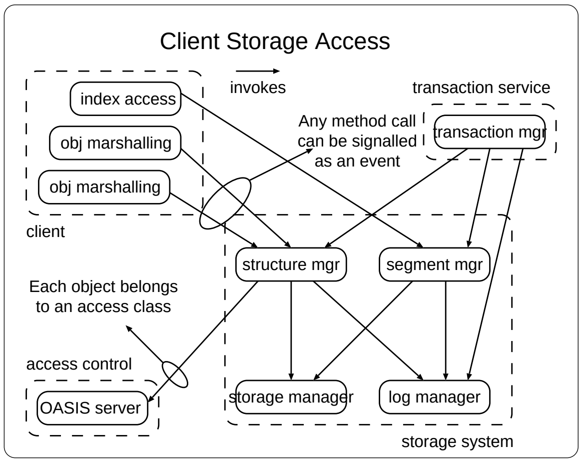

Fig. 7.7: Persistence Mechanism

7.5.1 Query Processing

The current implementation uses only the storage system of an ODMG DBMS, but it also provides the basic infrastructure for higher level services.

1. Queries will be processed by parsing the text form and returning a parse tree. The current system incorporates a generalised parsing system that handles any language for which a Backus-Naur-Form (BNF) grammar is supplied.

2. This parse tree is then handed to a module that performs type checking and type inference on it. The types of the most elementary sub-queries are determined by interrogation of the schema. Again, the current system provides the schema interface and already correctly returns all information necessary to complete this task.

3. When the annotated parse tree has passed the type validation stage, it will be optimised. Optimisation based on algebraic manipulation can be carried out without additional information, but optimisation based on the structure and statistics of the data will again require access to the database schema. The schema functions correctly, but may need extension to include information that would improve optimisation.

4. Finally, with an imperative query execution plan, a graph of operators and object streams can be constructed and executed. Inside the database, objects will not be instantiated, but rather their state will be inspected directly. This leaves two design decisions: returning of anonymous structures and invocation of object methods. These two issues will be addressed in this section.

Object Method Invocation

Queries may contain method calls, such as:

```
select x.age() from x as persons;
```

This will require the database management system to iterate over the complete extent `persons` and create an instance of each object in it. Each object will temporarily be named `x` and the `age()` method will need to be invoked on it.

This poses the problem of invoking code written by a programmer long after the database management system executable program has been compiled and linked. The method presented here operates as follows:

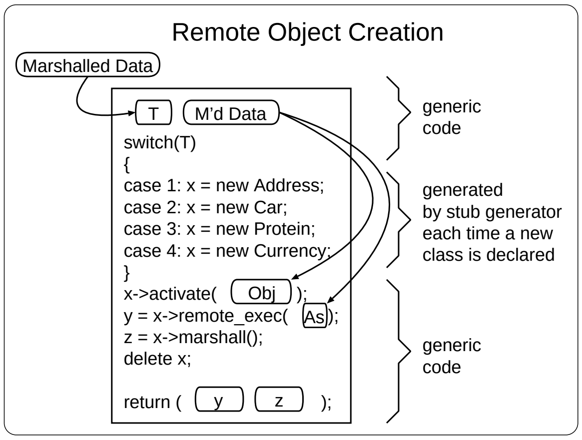

Fig. 7.8: Remote Object Creation

- The DBMS contains a generic method `remote_execute`, which takes as an argument the following tuple: [object state, method number, arguments] and returns [new object state, result].
- Upon execution of the method `remote_execute`, the DBMS first flushes all modifications into the database; this is a necessary measure since the user code may access arbitrary parts of the database and must be presented with a consistent view from inside the current transaction. Future enhancements to the ODL may specify that a method does or does not access other parts of the database and those annotations can then be used to optimise this rather drastic behaviour of flushing all state.
- The `remote_execute` method then marshals its tuple and the transaction identifier using a portable marshaling mechanism[3] and ships it off to a client program that has advertised itself to the database as being able to perform the method in question. This client program really is a computing server with the user-supplied method code linked in. Due to its transient nature, it should really be located through a trading mechanism outside the DBMS. Before trusting this outside service with its data, the DBMS ought to establish whether this service is to be trusted with the particular data set. In practice, it is to be expected that a compute server is completely included in the trust perimeter of the database itself. In this way, no further authorisation is necessary before handing the data to the remote compute server.

[3] Each structure can marshal itself given a generic container. Different container implementations allow for partial or system-dependent marshaling. Portable marshaling is more costly in terms of CPU cycles and memory used.

- Inside the computing server, the data is unpacked, an object instantiated (as shown in figure 7.8), the method is called (as shown in figure 7.9) and the resulting return value(s) along with the new object state marshaled again. The infrastructure for this procedure is provided by the stub generator and already functions correctly.
- Upon returning to the DBMS, the new object state is made current and the result values are used as prescribed by the query execution plan.

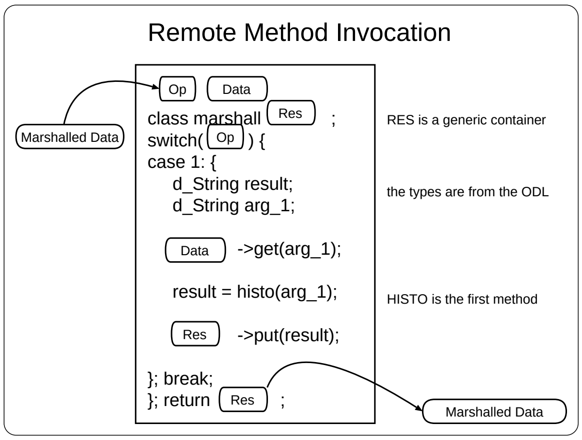

Fig. 7.9: Object Remote Execution

The database object inside the remote computing server contains the code to construct the most specific object type inside the given programming language. This code needs to be generated from the schema each time a new class declaration is added to the database.

Each class in the database is derived from the basic type `d_Object`. In our implementation, this type contains a method called `remote_exec` that takes a marshaled version of the method number and the required arguments and returns the marshaled results. This method is automatically generated and placed inside the stubs at the same time as the marshaling and unmarshaling code that is needed to provide persistence. The operation of this method is illustrated in figure 7.9.

The database schema contains enough information to produce this code and later on, the packet to operate this code can again be produced automatically by consultation of the database schema.

7.5.2 Distributed Transaction Manager

The transaction manager in this experimental setup is intended to conform to the OSI transaction processing standard. As described in section 2.4.2, each site is under autonomous control and a distributed protocol (the trans-address-space (TX) protocol) ensures that distributed transactions retain the ACID properties. Figure 7.10 shows a schematic overview of its interaction with the remaining system.

The concurrency control strategy used in this implementation is pessimistic (locking with deadlock detection).

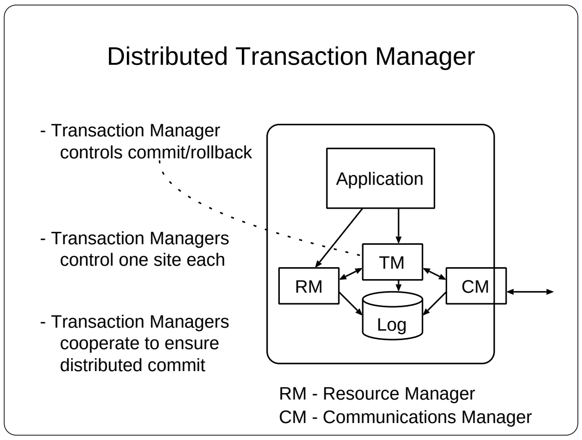

Fig. 7.10: Transaction Manager

7.6 Access Control

The access control mechanism envisioned for this system is based on the Open Architecture for Interworking of Secure Services (OASIS), as described in section 6.2. Figure 7.11 gives an overview of its features and operation.

The intended setup would include:

- authentication servers
- OASIS servers
- event servers

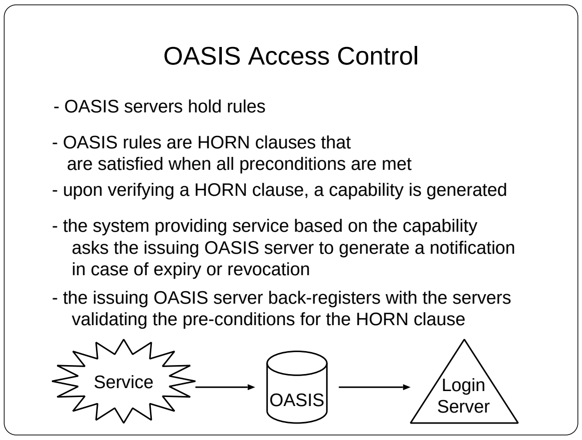

Fig. 7.11: Access Control Mechanism

The system would be operated by having the event servers (sources, proxies, relays, CESs, schema servers) trust some particular OASIS server. Each event server would have a set of rules that relate a role to a set of resources and access rights. The roles are defined in the OASIS server. If a particular client could prove membership in the role, that client would be able to perform the permitted operations on that resource. An end-user client could enter a particular role by satisfying one of a defining set of rules stored in the OASIS server. This is the computationally intensive part, since it involves proving a theorem, or less generally, finding some path through the rule graph in the OASIS server. Once the client has had this membership granted, the client is issued with a capability stating this membership.

8. CASE STUDIES

The central thesis of this dissertation consists of the belief that the value added by meta-data easily justifies the expense of obtaining, managing, and keeping it. This chapter presents a range of different applications for asynchronous messaging and explains how each of those applications benefits from having a machine-readable description of the types, configuration and state of the network available.

8.1 Network Monitoring

Event technology is used in the telecommunications industry, where the status of equipment is signalled to a centralised management facility. Ideally, each device in the network conforms to the event signalling protocol and is configured to send status messages to a concentrator that is upstream, towards the management facility.

At the lowest level, communications equipment can talk to each other already. Signalling of events is realised in-band on the same communications medium as the primary workload. The protocols for the equipment allow for multiplexing and the status and configuration messages are sent in their private messages or in their private virtual circuit. The infrastructure for those messages does not need to have much knowledge of their structure in order to forward them: it is enough to be able to indicate a destination, even if it is in form of a 'type=*location*', which would denote the very general destination 'everybody who is interested in *location* events'.

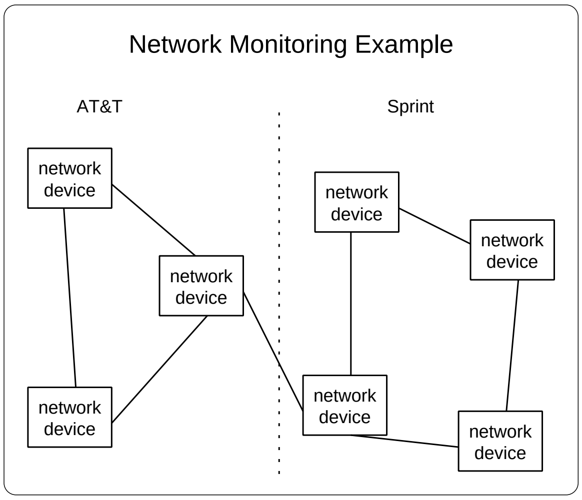

Fig. 8.1: Telecommunications Network-Monitoring

If the infrastructure can interpret more of the event, such as more fine-grained routing information, or priorities, or timestamps, then forwarding can happen with less processing and greater efficiency.

Problems with this approach are that not every element in the network is 'active' and may send signals, and that some faults may cause signals generated by other components to be lost. Passive network components can still be provided with designated monitoring equipment, that builds a wrapper around the passive element and presents an active interface to the event network.

The example depicted in figure 8.1 shows how the networks of two telecommunications providers are joined. The separate management domains make it difficult for either of the two parties to detect a fault at the joining connection.

If events are used to signal faults, then event expressions may be constructed that filter out only those events that are relevant to border-equipment. It is then easy to grant access to this particular stream of events. In this way, separate providers can guard the internal structure and operation of their network, while releasing exactly the information necessary to perform management at the border.

The problems arising from this example are those of separate management domains, as addressed in 5.1, and those of long-term changes to the event types and how this evolution can be handled best in practice. Chapter 6.1 offers guidance on how to formalise and automate this evolution process.

8.2 Workflow Coordination

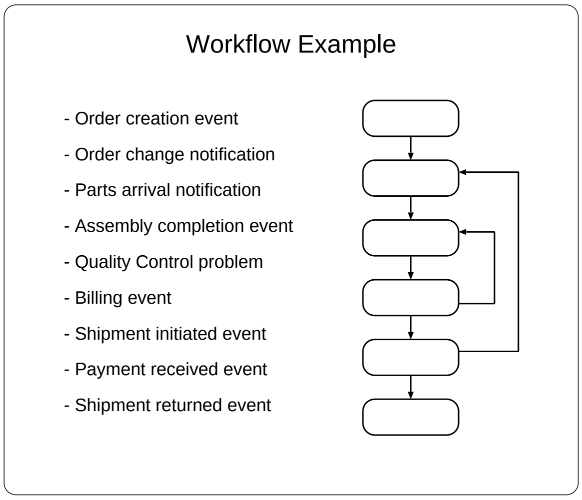

Fig. 8.2: Workflow Coordination

Another interesting application for event technology is workflow coordination. Figure 8.2 shows how a network of rules and conditions. When a certain goal is given and established as an independent process, it traverses this graph and passes through several stages. External occurrences cause the creation of this process and then advance it further.

Depending on the type of workflow, it may be linear, or it may be allowed to traverse a more complex graph, using external events to advance it and make decisions on how to proceed.

The use of event technology allows the control flow to be separated from the actual processing tasks. Each component in the process has to be engineered to allow processing of the work unit in accord with the instructions from an external event. This asynchronous mode of operation increases robustness, since the code is constructed more carefully to be independent from surrounding infrastructure. It also decreases computing load, since no effort is spent tying up resources while waiting for other components.

The benefits in this model arise from the compartmentalisation of tasks, so that they become replaceable, comprehensible and manageable. Components can be built easily for very specific purposes. More general components, that are intended for re-use can be designed to consume arbitrary events, relying on the infrastructure for information on the meaning of these events.

An added benefit of using publish-register technology for workflow coordination is that the information can be recorded non-intrusively and can then be mined using modern database techniques. This allows the effectiveness of processes to be analysed and judged.

8.3 Multi-Media Presentation and Collaborative Working

Events are often used in multi-media applications to achieve synchronisation. The example presented here allows streams from several sources to be joined, filtered, stored, replayed and annotated.

Applications for collaborative working are increasingly using multi-media data to let participants interact. The most frequently cited examples of multi-media are sound and movies (as shown in figure 8.3), where events are used to mark synchronisation points. But it is also conceivable to operate animated presentation packages, drawing packages, or word processors collaboratively, where each interaction produces an event and forwards it to all the involved parties.

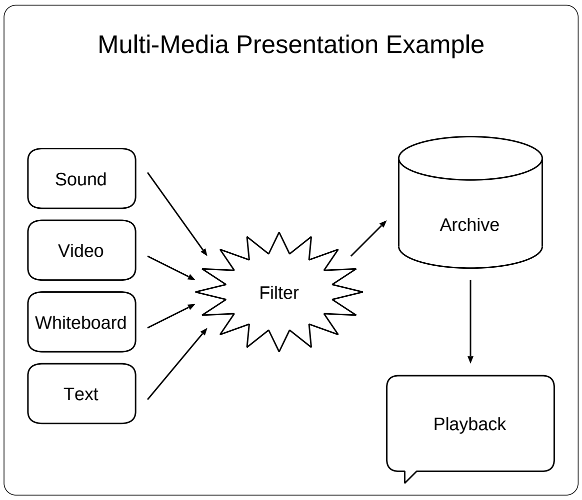

Fig. 8.3: Multi-Media Presentation

If the amount of data is small, like in a word-processor or vector drawing package, then the events can be used to represent all the information of the update. In this case, loss of connectivity can be recovered from. The strategy is to discard the parts of continuous media, such as speech and video, that were transmitted during the loss, and resynchronise these streams at reconnection time. The events that embody changes to a document, drawing or presentation, are buffered during times of lost connectivity and replayed when the connection exists again.

In this world, the metadata describes the capabilities of the sources and sinks. The event type hierarchy is constructed in such a way that the most detailed events contain the largest amounts of data and can only be processed by the most capable machines. Classification of machines according to the bandwidth at which they are connected, the size of the display and the amount of CPU cycles and memory, allows the appropriate data to be delivered to the destination. In this way, mobile computer, home machines, office computers and special multi-media laboratories can all be provided with the appropriate data. Utilising the discovery methods described in chapter 5.2, this translation and adaptation even happens automatically.

8.4 Digital Libraries

Digital libraries can make use of event technology in several areas: the OASIS access control mechanism (see chapter 6.2) relies on event technology to efficiently broadcast changes in access permissions. As digital libraries develop, they may not only acquire more content, but they may also enter into agreements with other libraries to permit reader access across boundaries. This development of the readership into a larger and larger group still needs to be controlled, and possibly different levels of service may be offered.

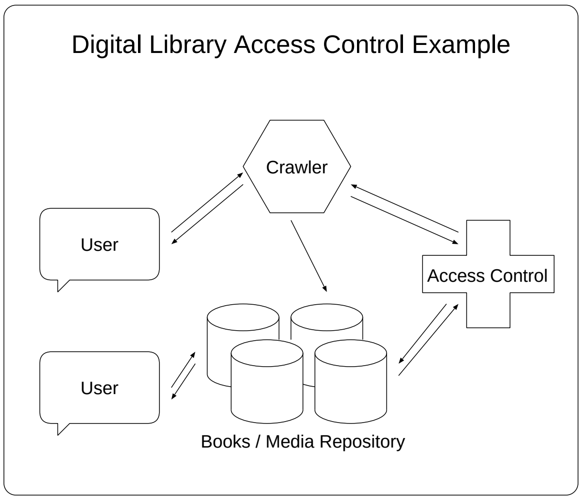

Fig. 8.4: Digital Library Access Control

A further use for event technology within a digital library is the asynchronous notification of research results. A digital library has a very large archive (compare figure 8.4), which is most likely not online. A full search across the archive is going to be performed periodically, with several queries being performed against each retrieved item. When an item matches the desired search characteristics, it is cached, and the user is notified. A user could choose to instantiate an agent that performs more elaborate checking, and possibly could produce a summary. In this situation, the agent is again going to benefit from having as much information as possible: which search produced the result, metadata about the retrieved item, such as age (currency), reliability and category. A further source of information might be the description of the event indicating the find. If the event came from a remote domain, the semantics for searches may not be the same as locally, so it is helpful to have machine-readable schemas to interpret this event.

9. CONCLUSION

This dissertation has presented a way of storing and organising type schemas and other metadata for the restricted application domain of open event stream systems. This chapter summarises the main conclusions, and suggests further work.

9.1 Summary

Existing event stream systems lack the supporting infrastructure to truly exploit their potential in the wide area and over time. In this dissertation, consideration was given to the provision of such discovery and selection services.

The review of current middleware technology (chapter 2) has given an overview of asynchronous messaging, directory implementations, transaction services and database connectivity. These are the foundation on which the remaining work is based and on which it expands. This review has in particular shown how the different event systems are converging in their functionality. Integration of these messaging systems with metadata repositories has not yet happened. The need for such integration is presented as part of this thesis.

The technology chosen for representing events is the ODMG standard. This framework defines an object model and recognises most of the problems found in schema federation and evolution. Chapter 3 has described the development history of the ODMG proposal and justifies the choice as a basis for the CEA event messaging framework. The ODMG proposal is useful because it gives a clear definition for the object state representation, and offers a rich interface for access to the database schema.

Chapter 4 has proposed a design for using the ODMG metadata specification as the basis for an event schema repository and an event services broker. It has shown how events can be classified, which properties they have and how these features can be represented within the ODMG Metadata framework. Beyond event properties, the chapter elaborates on how event services can be described, classified, stored and accessed through the ODMG Metadata interface. As a specific case, the representation of gateways is discussed and it is shown how a more specific service type can be represented with greater semantic accuracy. The design for storing all of this data provides the basis for using it in the subsequent chapters.

The chapter on Wide-Area Event Services describes the difficulties arising when autonomous domains are interconnected. It shows, how access rights, services types and directory functionality suffer from this interconnection. As a deeper problem, the mismatch of heterogeneous event technology is highlighted and some of the resulting mismatches are explained. The chapter concludes with a recommendation, on how domains can be connected, how the infrastructure can be made available across boundaries, and how each domain can still remain autonomous and be protected against unauthorised access and resource overuse. This scheme relies on the availability of event and service types, user identifiers, the configuration of channels and filters, a history of the schema evolution and named service entry points.

Chapter 6 is set against the background of a repository design and an event system interworking scheme. It has described in detail, how the evolution of a distributed system can be managed in a single domain. Chapter 6 then explained, how this management can be extended to several domains and how domain autonomy can be retained during this process. Finally it described, how the restricted application of an event system allows for evolution and interworking, and can to a large degree be managed automatically.

Chapter 7 describes the experimental setup that has been used to demonstrate the use of metadata. After a discussion of successful distributed technologies, a motivating example set the scene for further investigation. A portrait of the basic event infrastructure, the composite event server and the gateway represented the messaging section. A description of the schema repository, the trader and the search engine showed how the metadata relating to this infrastructure is stored. The section on persistence covered in detail the design for query processing, object movement across protection domains and through time, and the distributed transaction manager. It is this chapter which showed how design for an anonymous and asynchronous model considers more modes of failure and leads to more robust applications than result from synchronous point-to-point methods. It also showed how the separation of control flow into the framework and computation in components leads to improved flexibility over closed systems.

Case studies for this technology were presented in chapter 8. Manageability is of prime importance in the first example, a telecommunications network monitoring application. In this example, the availability of service descriptions allows for very flexible and select disclosure of information. The portrait of a workflow application showed how the increased flexibility can be used to adapt a system rapidly to new market conditions and how the available descriptions allow better mining of the accumulated data. A multi-media application was used as an example to illustrate how event technology can be used to increase robustness. The case portrayed was that of a network disconnection and of event buffering to prevent data loss. It was again the availability of detailed metadata that allowed the integration of diverse sources into a flexible application. The final example is that of a federation of digital libraries. It is here again that manageability is of prime importance, as cross-licensing and resource restrictions are to be implemented.

9.2 Suggestions For Further Work

As a setting for further work, the active home offers a rich test-bed. The active home is an experiment of incorporating more communications technology into everyday life. Figure 9.1 illustrates some of the sensors and actuators that can be integrated into this system. It is intended to connect all of these sensors and actuators to a network. With such a pervasive architecture, stringent access controls need to be employed. It will also be necessary to employ a framework that allows for easy addition of components and that permits the extension of the services. The management requirements for this environment will demand a transparent and informative interface. The architecture presented in this dissertation could be tested in this real-world situation and its usefulness evaluated.

Many of the services in the active home will be operated through simple feedback-control. It is expected that the composite event service will be able to provide most of this functionality:

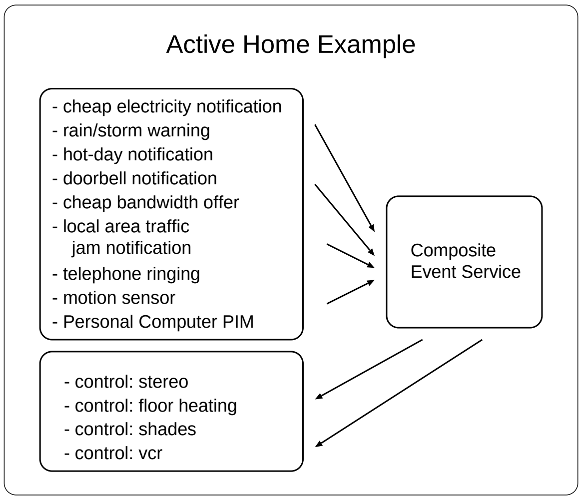

Fig. 9.1: Active Home

The diagram above shows several event sources that can be used to provide the input for control decisions in the active home. Many of the issues surrounding federation need testing out and extending the active home to be a single management domain, with essential community services constituting further domains, would provide an ideal test-bed for such experiments.

A further possible extension of this work would be to investigate how the methods of metadata collection, storage, and use can be applied to the domain of applications beyond event technology.

9.3 Conclusion

The prototype of the metadata storage service has shown that it is possible to collect and maintain meta information ranging from static type hierarchies to dynamic registration information. Based on this experience, a framework for graceful evolution and federation of distributed applications has been proposed. Extensive consideration has been given to the interworking of event systems with mis-matched capabilities. This work has shown that the collection of metadata on the capabilities and configuration of these systems can lessen the impact of this problem.

APPENDIX

A. LIST OF TERMS AND ABBREVIATIONS

2PC	2-Phase Commit (see 2.4.2)
ACL	Access Control List
API	Application Program Interface
basis	A function maps from an element with type A to an element of type B. The type A is called the basis, while type B is called the domain.
BNF	Backus-Naur-Form for a formal description of language syntax [Bac78]
BSC	Byte Segment Custode (see [Lo94])
CDS	*Cell Directory Service* (see 2.3.8)
CEA	Cambridge Event Architecture (see 2.2.3)
CES	Composite Event Service (see 7.2)
COM	Microsoft Component Object Model (see 2.1.4)
CORBA	*OMG Common Object Request Broker Architecture* (see 2.1.4)
COS	*CORBA* Common Object Services [Obj98b]
data independence	A concise description of *data independence* can be found in Graham Kemp's [Kem95]: Data independence is defined by Date ([Dat90], second edition 1977) as "immunity of applications to change in storage structure and access strategy". The ANSI-SPARC three layer model [Ins75] demanded that database systems be constructed so that they provided both logical and physical data independence. Logical data independence provides that the conceptual schema must be able to evolve without changing external application programs, while physical data independence allows one to substitute a different physical storage schema for an existing one, and to reload the data into the new form of storage, without any change either to the conceptual schema or application programs.
DCE	Open Software Foundation *Distributed Computing Environment* (see 2.1.4)
DCOM	Microsoft Distributed COM (see 2.1.4)
DDL	Data Definition Language (see 3.2)
DEC	Digital Equipment Corporation, bought by Compaq Computer Corporation, P.O. Box 692000, Houston, TX 77269-2000, U.S.A.
DFS	*Distributed File Service* (see 2.1.4)
DIB	*X.500* Directory Information Base (see 2.3.3)
DIT	*X.500* Directory Information Tree (see 2.3.3)
DML	Data Manipulation Language (see 3.2)
DQL	Data Query Language (see 3.2)
DTD	SGML/XML Document Type Definition (see 2.5.5)
DTP	*Distributed Transaction Protocol* (see 2.4.2)
DTS	*Distributed Time Service* (see 2.1.4)

ECO	Events, Constraints and Objects, (see 2.2.3)
EDI	Electronic Data Interchange (see 2.2.1)
email	This stands for *electronic mail*, but people like Professor Donald Knuth have convincingly argued that by now this shortcut should be considered a word of its own.
FDM	Functional Data Model (see 3.1.4)
FEC	Forward Error Correction (Coding)
GDS	*Global Directory Service* (see 2.3.8)
GEM	Generalised Event Monitoring language (see 2.2.3)
HTML	Hyper Text Mark-up Language (see 2.5.5)
HTTP	Hyper Text Transfer Protocol (see [FGM+99])
IBM	International Business Machines
IDL	Interface Definition Language
ISO	International Standards Organisation, Geneva
ITU	International Telecommunications Union, Geneva, Switzerland
JDBC	Java Database Connectivity (see 2.5.3)
JINI	The JINI FAQ list on Sun Microsystems' WWW site states that JINI is not an acronym, but is spoken in the same way as 'genie', making an allusion to the magical powers of this technology.
JPMQ	*Java* Persistent Message Queue
JVM	*Java* Virtual Machine
LDAP	Light-Weight Directory Access Protocol (see 2.3.4)
LHS	Left Hand Side
MOM	Message Oriented Middleware (see 2.2.1)
MOTD	Message Of The Day
MSSA	Multi Service Storage Architecture (see 2.1.1)
MTS	Microsoft Transaction Server (see 2.4.6)
NDS	*Novell Directory Services* (see 2.3.5)
NIS	Network Information Service (see 2.3.2)
ODBC	Open Database Connectivity (see 2.5.1)
OID	Object Identifier (see 2.1.2)
OLTP	On-Line Transaction Processing (see 2.4.5)
OMG	Object Management Group, OMG Headquarters, Framingham, MA, U.S.A. (see 2.1.3)
ontology	An ontology is a classification graph with semantic descriptions of the capabilities of objects belonging to the class of a particular node. The semantic descriptions are typically recorded in terms of typed attributes, methods, relationships and constraints.
OODBMS	Object Oriented Database Management Systems (see 3)
ORB	Object Request Broker (see 2.1.4)
OSF	Open Software Foundation, X/Open Ltd., Reading, Berkshire, England (see 2.1.3)
OSI	*Open System Interconnect* [Org95]

OTS	*CORBA* Object Transaction Service (see 2.4.4)
PDF	Adobe Portable Document Format [BC93]
polymorphism	*Parametric polymorphism* or *function polymorphism* allows a particular function to accept arguments of differing types, such as objects from different object classes. *Ad-hoc polymorphism* is a programming language property in which one function (say *operator+*) may be used on different data types (on integers, floats and strings). *Data polymorphism* allows a template to be constructed for a particular collection of data types (for example a generic list type).
QoS	Quality of Service (see 2)
RDBMS	Relational Database Management System
RDF	XML Resource Description Framework (see 2.5.5)
RM-ODP	*Reference Model for Open Distributed Processing* (see 2.1.3)
RMI	(*Java*) Remote Method Invocation (see 2.1.4)
RPC	Remote Procedure Call
SFC	Structured file Custode (see [Tho90])
SGML	Standard Generalised Mark-up Language (see 2.5.5)
SQL	Structured Query Language [ISO92, CAB+81]
TCB	Trusted Computing Base
TCP	Transmission Control Protocol [Pos81]
TINA	*Telecommunications Information Networking Architecture* (see 2.1.3)
TIP	Transaction Internet Protocol (see 2.4.3)
URL	Uniform Resource Locator [Moa97]
W3C	World Wide Web Consortium (see 2.5.5)
WWW	World Wide Web
X/Open	Prefix for products of the OSF (see 2.1.3)
XFN	X/Open Federated Naming (see 2.3.8)
XML	eXtensible Mark-up Language [BPSM99]
XSL	eXtensible Stylesheet Language (see 2.5.5)
XSLT	XSL Transformations (see 2.5.5)

BIBLIOGRAPHY

[AHHR+93] K. Ahmad, P. Holmes-Higgin, M. Rogers, M. Hoge, K. Le Hong, C. Huwig, R. Kese, and R. Mayer. User-driven software development: Translator's workbench – an exemplar case study. In *Proceedings of the Fifth International Conference on Human-Computer Interaction*, volume 1 of *II. Special Applications*, pages 319–324, 1993.

[AJ99] M. Atkinson and M. Jordan. Issues raised by three years of developing PJama: An orthogonally persistent, platform for Java[TM]. *Lecture Notes in Computer Science*, 1540:1–30, 1999.

[Atk88] M. P. Atkinson. Ps-algol reference manual, 4th edition. Technical report, Universities of Glasgow and St Andrews, 1988.

[Atk89] Atkinson. The object oriented database system manifesto. In *Proceedings of the 1st International Conference on Distributed and Object-Oriented Design*, International Conference on Distributed and Object-Oriented Design. Elsevier Science Publishers, Kyoto, Japan, 1989.

[Bac78] J. Backus. Can programming be liberated from the von-neuman style? *Communications of the ACM*, 21(8):613–641, 1978.

[Bat94] John Bates. Interactive multi-media presentation. PhD dissertation 231, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, August 1994.

[BBHM95] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using events to build distributed applications. In *Second International Workshop on Services in Distributed and Networked Environments*, pages 148–155. IEEE Computer Society Technical Committee on Distributed Processing, IEEE Computer Society Press, June 1995.

[BC93] Tim Bienz and Richard Cohn. *Portable Document Format Reference Manual.* Addison-Wesley Publishing Company, July 1993.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. Asset: A system for supporting extended transactions. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):44–54, June 1994.

[BFKM85] Lee Brownston, Robert Farrel, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5, An Introduction to Rule-Based Pro-*

gramming. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985.

[BHL99] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in xml. http:// www.w3.org/ TR/ REC-xml-names, January 1999.

[BK] N. Brown and C. Kindel. Distributed component object model protocol – dcom/1.0. http:// www.microsoft.com/ oledev/ olecom/ draft-brown-dcom-v1-spec-01.txt.

[BM91] Catriel Beeri and Tova Milo. A model for active object oriented database. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, number 17 in VLDB, pages 337–349, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, September 1991. Morgan Kaufmann Publishers, Inc.

[BPSM99] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. http:// www.w3.org/ TR/ REC-xml, February 1999.

[BS96] Inc. BEA Systems. Bea tuxedo whitepaper. http:// www.beasys.com/ products/ tuxedo/ tuxwp_pm/ tuxwp_pm1.htm, November 1996.

[C4496] Data management: Structured query language (sql) version 2 (call level interface), March 1996.

[CAB+81] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A History and Evaluation of System R. *Communications of the ACM*, 24(10):632–646, 1981.

[Cat97] Rick G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 2.0 edition, 1997.

[CD99] James Clark and Steve DeRose. Xml path language (xpath). http:// www.w3.org/ TR/ xpath, July 1999.

[Cla99] James Clark. Xsl transformations (xslt). http:// www.w3.org/ TR/ xslt, October 1999.

[Col97] R. M. Colomb. Impact of semantic heterogeneity on federating databases. *The Computer Journal*, 40(5), 1997.

[Con94] TINA Consortium. Tina-dpe services. Technical report, TINA Consortium, December 1994.

[DA99] Stephen J. DeRose and Paula Angerstein. Xml xlink requirements. http:// www.w3.org/ TR/ NOTE-xlink-req, February 1999.

[Dat90] C. J. Date. *An Introduction to Database Systems*, volume I. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 5 edition, 1990.

[Dea99] Stephen Deach. Extensible stylesheet language (xsl) specification. http:// www.w3.org/ TR/ WD-xsl, April 1999.

[DeR99] Steven J. DeRose. Xml xpointer requirements. http:// www.w3.org/ TR/ NOTE-xptr-req, February 1999.

[(Ed98] Michael J. Greenberg (Editor). *Notification Service*. Object Management Group, OMG Headquarters, Framingham, MA, U.S.A., omg 98-06-15 edition, June 1998.

[Emb95] Suzanne Embury. *The P/FDM Object Oriented Database*. Object Database Group, Dept. of Computing Science, University of Aberdeen, UK, February 1995.

[FCHH99] Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC(TM) API Tutorial and Reference*. Addison-Wesley Publishing Company, 2 edition, June 1999.

[Fer97] William Ferreira. A stream-based semantics for composite events. available within the UoC OPERA research group, August 19 1997.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. World Wide Web Consortium, June 1999.

[Fly99] Peter Flynn. The extensible markup language – frequently asked questions v1.5. http:// www.ucc.ie/ xml/ index.html, June 1999.

[Gel85] David Gelernter. Generative communication in linda. *TOPLAS*, 7(1):80–112, January 1985.

[Gol86] Charles F. Goldfarb. *SGML: Standard Generalized Markup Language*. ISO/IEC JTC1/SC34/WG1, Dr. James D. Mason, Lockheed Martin Energy Systems, Information Management Services, 1060 Commerce Park, M.S. 6480, Oak Ridge, TN 37831-6480 U.S.A., 1986.

[GR91] Charles F. Goldfarb and Yuri Rubinski. *The SGML Handbook*. Clarendon Press, February 1991.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1993.

[Gro91] The Open Group. The x/open standard for distributed transaction processing (dtp), 1991. X/Open Common Application Environment.

[Hay96] Richard Hayton. Oasis: An open architecture for secure interworking services. Technical report, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, March 1996.

[Hul97] R. Hull. Managing semantic heterogeneity in databases: a theoretical prospective. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 51–61, New York, NY 10036, USA, 1997. ACM Press.

[Ins75] American National Standards Institute. Interim report of the ansi/x3/sparc study group on data base management systems. *ACM SIGFIDET*, 7:3–139, 1975.

[ISO92] International standard for the structured query language (sql), 1992.

[ITU95] Odp reference model part1: Overview. Draft International Standard, 1995.

[KA86] K. G. Kulkarni and M. P. Atkinson. Efdm: Extended functional data model. *The Computer Journal*, 29(1):38–46, February 1986.

[Kem95] Graham J. L. Kemp. A functional data model approach to integrating biological databases. In *Proceedings of Meeting on the Interconnection of Molecular Biology Databases*, Cambridge, UK, July 20-22 1995. Department of Computer Science, University of Aberdeen, King's College, Aberdeen, UK, AB9 2UE.

[LEK98] J. Lyon, K. Evans, and J. Klein. *Transaction Internet Protocol V1.0.* Internet Task Force, July 1998.

[LN89] R. J. Lipton and J. F. Naughton. Estimating the size of generalized transitive closures. In *Proceedings of the 15th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Amsterdam*, August 1989.

[Lo94] Sai Lai Lo. A modular and extensible network storage architecture. Technical report, University of Cambridge Computer Laboratory, January 1994.

[LS99] Ora Lassila and Ralph R. Swick. Resource description framework (rdf) model and syntax specification. http:// www.w3.org/ TR/ REC-rdf-syntax, February 1999.

[Mac98] International Business Machines. Transarc encina overview. http:// www.software.ibm.com/ ts/ txseries/ library/ techsheets/ encina/ index.html, January 1998.

[Mar95] Elizabeth A. Martin. X/open federated naming. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 46(6):28–33, December 1995.

[Mau99] Jon Mauney. Osf dce frequently asked questions. http:// www.ivc.com/ mauney/ dce-faq.html, March 1999.

[MB98] Chaoying Ma and Jean Bacon. Cobea: A corba-based event architecture. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems COOTS-98)*, pages 117–132, Berkeley, April27–30 1998. USENIX Association.

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

[MDK94a] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5):304–312, September 1994.

[MDK94b] Jeff Magee, Naranker Dulay, and Jeff Kramer. A constructive development environment for parallel and distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5):304–312, September 1994.

[Moa97] R. Moats. *Unversal Resource Names*. Internet Task Force, May 1997.

[Moc83] P. Mockapetris. *Domain Names – Concepts and Facilities*. Internet Engineering Task Force, November 1983.

[MSS95] M. Mansouri-Samani and M. Sloman. Gem: A generalised event monitoring language for distributed systems. Technical report, Imperial College Department of Computing, Department of Computing, Imperial College, London, UK, 1995.

[Net] Micro Soft Developers Network. Open database connectivity. http:// www.microsoft.com/ Data/ odbc/ faq_odbc.htm.

[Net97] Micro Soft Developers Network. Internet explorer active channels overview. http:// msdn.microsoft.com/ workshop/ delivery/ channel/ overview/ overview.asp, 1997.

[Obj97] Object Management Group, OMG Headquarters, Framingham, MA, U.S.A. *The CORBA Object Transaction Service*, omg 97-12-17 edition, November 1997.

[Obj98a] Object Management Group, OMG Headquarters, Framingham, MA, U.S.A. *The Common Object Request Broker: Architecture and Specification, revision 2.2*, omg 98-02-33 edition, February 1998.

[Obj98b] Object Management Group, OMG Headquarters, Framingham, MA, U.S.A. *Common Object Services Specification Volume 1*, omg 98-02-33 edition, February 1998.

[OC96] Karl O'Connell and Vinny Cahill. System support for scalable distributed virtual worlds. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 141–142. ACM, July 1996.

[OCDC+96] K. O Connell, T. Dinneen, S. Collins, B. Tangney, N. Harris, and V. Cahill. Techniques for handling scale & distribution in virtual worlds. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, August 1996. Association for Computing Machinery.

[Org89] International Standards Organisation. Open system interconnect – commit, concurrency control and recovery, 1989.

[Org92] International Standards Organisation. Open system interconnect – distributed transaction processing, 1992.

[Org95] International Standards Organisation. Open system interconnect – reference model, June 1995.

[PAD+97] Tony Printezis, Malcolm P. Atkinson, Laurent Daynès, Susan Spence, and Pete Bailey. The design of a new persistent object store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997. To be published.

[Pos81] Jon Postel. *Transmission Control Protocol*. DARPA, September 1981.

[Rad96] Elke Radeke. Extending odmg for federated database systems. In *DEXA Workshop*, pages 304–312, 1996.

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

[RHJ98] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Html 4.0 specification, April 1998.

[Rog97] Dale Rogerson. *Inside COM (Programming Series)*. Microsoft Press, February 1997.

[Ros93] R. Rosenbaum. *Using the Domain Name System To Store Arbitrary String Attributes*. Internet Engineering Task Force, May 1993.

[Row98] A. Rowstron. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web*, 1(3):167–179, 1998.

[SB98] M. D. Spiteri and J. Bates. An architecture to support storage and retrieval of events. In Nigel Davies, Kerry Raymond, and Jochen Seitz, editors, *Middleware 98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware, pages 443–458, www.springer.co.uk, September 1998. IFIP, Springer-Verlag London Limited.

[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, December 1995. Springer-Verlag. Also technical report TCD-CS-95-28, Dept. of Computer Science, Trinity College Dublin.

[Sha89] Y-P. Shan. An event driven model-view-controller framework for smalltalk. In *Object-Oriented Programming Systems Languages and Applications*, OOPSLA, pages 347–352, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, October 1989. Morgan Kaufmann Publishers, Inc.

[She91] Amit P. Sheth. Federated database systems for managing distributed, heterogeneous, and autonomous databases. In *International Conference On Very Large Data Bases (VLDB '91)*, pages 489–490, Hove, East Sussex, UK, September 1991. Morgan Kaufmann Publishers, Inc.

[Shi81] D. W. Shipman. The functional data model and the data language daplex. *ACM Transactions On Database Systems*, 6:140–173, 1981.

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In USENIX Association, editor, *USENIX Conference Proceedings (Dallas, TX, USA)*, pages 191–202, Berkeley, CA, USA, Winter 1988. USENIX Association.

[SR99] Adrian Skehill and Mark Roantree. Constructing multidatabase collections using an extended odmg object model. *EFIS*, pages 95–106, 1999.

[Swi99] Ralph Swick. Metadata activity statement. http:// www.w3c.org/ Metadata/ Activity.html, October 1999.

[SZ98] Hans-Peter Steiert and J"urgen Zimmermann. Jpmq – an advanced persistent message queuing service. In *16th British National Conference on Databases*, BNCOD, Cardiff, 1998.

[TaSRL+90] Michael The Committee for Advanced DBMS Function are Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, and David Beech. Third-generation database system manifesto. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19 (3), pages 31–44, 1515 Broadway, New York, NY 10036, USA, 1990. Association for Computing Machinery, Association for Computing Machinery.

[Tho90] Sue Thomson. *A Storage Service For Structured Data*. PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, November 1990.

[Tho98] Anne Thomas. Enterprise javabeans technology whitepaper. http:// java.sun.com/ products/ ejb/ white_paper.html, December 1998. prepared for Sun Microsystems, Inc.

[W+98] J. Waldo et al. Javaspace specification - 1.0. Technical report, Sun Microsystems, March 1998.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *Ibm Systems J.*, 37:454–474, 1998.

[X.598] Itu-t recommendation x.500 / iso/iec 9594-1: The directory: Overview of concepts, models and services. International Standard, November 1998.