# State of the Art in Enablers for Applications in Future Mobile Wireless Internet

Fuego Core Project

Jaakko Kangasharju Tancred Lindholm
Kimmo Raatikainen Sasu Tarkoma

June 30, 2002

Helsinki Institute for Information Technology

# Contents

# List of Figures

# Chapter 1

# Introduction

One significant trend in software for future mobile systems is the requirement of ever-faster service development and deployment. An immediate implication has been the introduction of various service/application frameworks/platforms. Middleware is a widely used term to denote a set of generic services above the operating system. Although the term is popular, there is no consensus of a definition (see RFC2768). Typical middleware services include directory, trading and brokerage services for discovery, transactions, persistent repositories, and different transparencies such as location transparency and failure transparency. The importance of middleware, that is a set of generic services above operating system and transport stack, is widely recognized.

The objective of the Fuego Core project is to specify the set of fundamental enabling middleware services for mobile applications on future mobile environments and to implement two research prototypes. The project has adapted a two-level approach to develop the necessary middleware services. On the top-level, the *work areas* characterize the long-term vision in research for middleware for future mobile internet. On the bottom-level, the work areas are further split to *work items* that are addressed in the project.

The work areas in the Fuego Core include

**Adaptive Applications.** Adaptability is one of the key research areas in nomadic computing. The basic principle of adaptability is simple. When the circumstances change, then the behaviour of an application changes according to the desires of the user. Therefore, we need means to collect and to present user preferences, which may, in turn, depend on location, time, access device, properties of connectivity.

The basic principle of adaptability, i.e. the behaviour of an application changes when the circumstances change, requires that the system detects changes and notifies about them. Therefore, the generic service elements must include: *Environment Monitoring*, and *Event Notification*.

In environment monitoring there are three primary issues:

- discovery (which equipment are available),
- service location (which services are available), and
- available capabilities (computing power, various storage capabilities, available capacity on communication paths).

**Dynamic Reconfigurable Services.** Situations, in which a user moves with her end-device and uses information services, are challenging. Moreover, the nomadic user of tomorrow will not appreciate a static binding between her and an access device; not even in the case of multi-mode access devices that can handle several access technologies including wireless LAN, short-range radio, and packet radio. It must be possible to move a service session (or one end-point of a service session) from one device to another.

In these situations the partitioning of applications and the placement of different co-operating parts is a research challenge. The support system of a nomadic user must distribute, in an appropriate way, the parts among the end-user system, network elements and application servers. In addition, when the execution environment changes in an essential and persistent way, it may be beneficial to redistribute the co-operating parts. The redistribution or relocation as such is technically quite straightforward but not trivial. On the contrary, the set of rules that the detection of essential and persistent changes is based on is a challenging research issue.

In the dynamic configuration we have a huge space of research items. On the conceptual level there are research issues related to profiles, various kinds of context also including the social context, roles and trust. On the technical level we must solve the problems related to authentication, authorization, and delegation.

**Mobile Distributed Information Base** File and information synchronization between different devices is already available but in quite primitive forms. A single information base for a user-possibly different views for her different roles-and for multiple user groups is a fundamental enabler for seamless reconfiguration of the end-user system for a mobile user and for seamless user roaming from one role to another one.

The mobile distributed information base should provide consistent, efficiently accessible, reliable and highly available information base. This implies a distributed and replicated world-wide "file system" that also supports intelligent synchronization of data after disconnections. Shared access and support of transactional operations also belong to the list of requirements.

To summarize, the key enablers for mobile distributed information base include:

- distributed and replicated world-wide information storage that provides data consistency, efficient and reliable access and high availability,
- intelligent synchronization after disconnections, and
- distributed mobile transactions with flexible correctness criterion.

Of the fundamental enablers for future mobile applications, the Fuego Core project has selected three as the work items of 2002:

1. Event-based systems

   This work item addresses the fundamental principle of adaptability-the behaviour of an application changes when the circumstances change. Therefore, the system must detect changes and notifies about them. In other words, the middleware solution must provide service elements for Environment Monitoring and Event Notification.

2. XML issues on profile presentation, protocol, and transport over wireless

   XML starts to be the key presentation format for various kinds of information about capabilities, preferences, and properties. Therefore, middleware for mobile internet must provide an efficient way of exchanging XML content and of supporting SOAP.

3. Intelligent synchronization

   Of the fundamental enablers for mobile distributed information base intelligent synchronization is selected as the starting point. The assumption is that an existing storage system, for example Coda, OceanStore, or InterMezzo, can be integrated with syncML. The objective is to build mechanisms that take care of decisions on what and when to synchronize.

# Chapter 2

# Event-Based Systems

## 2.1 Introduction

This chapter presents an overview of event systems and distributed event frameworks with an emphasis on the special requirements presented by mobile computing. By mobile or ubiquitous computing we mean the new field of research created by wireless communication and the introduction of small, mobile devices. Traditionally, event-based systems are based on a number of event sources and event sinks, which register to receive certain type of events. Events are found everywhere; in Nature, office buildings and computer programs. An event-based framework can be decomposed into two essential parts:

- Event detection, which deals with the detection of the occurrence of a particular event of interest.
- Event notification, which is the act of notifying interested parties that an event has occurred.

Many existing platforms employ the synchronous model of method invocation, in which operations are performed on passive objects. This model is insufficient for reactive environments, where components need to react to changes, events, within the system and give timely response. An option would be to use polling the states of objects, but too frequent polling burdens the system and too infrequent polling delays the communication [BMH+00]. Asynchronous events support different applications types as identified by [BMH+00]:

- Group interaction
- Multimedia support (multimedia control through rules)
- Mobility

- Alarms and exceptions
- Management

Reliable and efficient asynchronous event detection and event notification are vital for the development of the next generation-distributed software for mobile Internet-aware devices. Event frameworks provide a plug-and-play architecture for creating distributed applications.

Distributed architectures are based on middleware that provides the interoperability layer required for heterogeneous cross-operating system and cross-language operation and communication. Components from different systems and different manufacturers can interoperate using middleware, such as CORBA, where the interface definitions created using the IDL (Interface Definition Language) can be shared. CORBA and Java provide the basis for more complex and interoperable software over various networked domains.

Currently middleware solutions, such as Java, from the desktop world are being introduced into the wireless world, where the requirements are different. Small and wireless devices have limited capabilities when compared with desktop systems: their memory, performance, battery life and connectivity are limited and constrained. The requirements of mobile computing need to be taken into account when designing an event framework that integrates with mobile devices. From the mobility and wireless viewpoint event systems can be divided into three distinct categories:

1. Traditional event systems designed for fixed network operation.
2. Event systems that support intermittent clients using a client-server protocol and possibly roaming between access nodes.
3. Ad hoc networks, where clients can also be servers and servers may roam.

The first category is the most researched and most of the architectures presented in this chapter fall into this category. Several architectures support intermittent clients and roaming between access nodes. Ad hoc event architectures are currently emerging, and they are only mentioned in this chapter.

From the small device point of view, message queuing is a frequently used communication method, because it supports disconnectedness. When a client is disconnected, messages are inserted into a queue and when a client reconnects the messages are sent. The distinction between the popular message queue based middleware and notification systems is that message queue based approaches are a form of directed communication, where the producers explicitly define the recipients. The recipients may be defined by the queue name or a channel name, and the messages are inserted into a named queue, where the recipient extracts messages.

Notification-based systems extend this model by adding an entity, the event service or event dispatcher, that brokers notifications between producers of information and subscribers of information. This undirected communication supported by the notification model is based on message passing and retains the benefits of message queuing. In undirected communication the publisher does not know, which parties receive the notification. This applies also to message-oriented middleware such as JMS [Sun01] that support publish-subscribe type of communication [SAS01].

Undirected communication decouples producers and consumers. In addition many systems support filtering and pattern detection that are used to reduce the amount of transmitted information and improve the accuracy of notifications. Content-based routing is flexible, because it does not require configuration information pertaining to channel names. Undirected communication may also be used to deliver the same set of information to a number of client devices. However, this requires associating user subscription information with a set of devices [SAS01] [CN01].

This chapter is structured as follows: Section 2.2 introduces event models, event routing and a number of requirements for mobile clients. Section 2.3 presents event systems such as the CORBA Notification Service, Siena, and Elvin. We examine the support for disconnected operation and mobility in each of the presented event systems. Finally, Section 2.4 presents the conclusion.

## 2.2 Event Models

Event models consist of event sources, event listeners, notification services, filtering services, and event storage and buffering services. In addition, there may be one or more authentication schemes to enforce security and access control. This section focuses on the general definition of events and event models.

### 2.2.1 Events

An event represents any discrete state transition that has occurred and is signalled from one entity to a number of other entities. For example, successful login to a service, the firing of detection or monitoring hardware and the detection of a missile in a tactical system are all events. An event may be based purely on software or it may be based on hardware. In addition, natural systems, such as biological cells, are also event-based systems. Nerve cells trigger impulses that travel around our body.

The firing of each event is either deterministic or probabilistic. A source can generate a signal every second making it deterministic. A stochastic source follows some probabilistic model that can be described using, for example, a Markov chain. Both event qualities can be modelled by building statistical or stochastic models of the firing behaviour of the event source. For example, a correlation analysis can be made between a series of event occurrences in time or between two event sources. Such an analysis would measure how strongly one event implies the other or how two event source firings are related.

Events may be categorized by their attributes, such as what physical property they relate to. For instance spatial events and temporal events denote physical activity. Moreover, an event may be a subset of basic event types, for example an event that has both a temporal and a spatial aspect.

Events can be categorized into taxonomies on their type and complexity. More complex events, called compound events, can be built on more specific simple events. Compound events are important in many applications. For example, a particular compound event can be fired when:

- In a hospital, when the reading of a sensor attached to a patient exceeds a given threshold and a new drug has been administered in a given time interval.
- In a location tracking service, where a set of users is in the same room or near the same location at the same time.
- In an office building, where a motion detector fires and there has been a certain interval of time after the last security round.

Event-based interaction can be:

- Discrete.
- Continuous, as event streams.

Events can also have different prioritisations, and event aging assigns an expiry time to each event notification. Event expiring prevents the spreading of obsolete information.

### 2.2.2 Event Model

The standard client/server communication models in distributed object computing are based on synchronous method invocations. For example, COM+, Java RMI and CORBA use synchronous calls (CORBA 3.0 supports asynchronous invocations). This approach has several limitations [GCSO01]:

- Tight coupling of client and server lifetimes. The server must be available to process a request. If a request fails the client receives an exception.

- Synchronous communication. A client must wait until the server finishes the processing and returns the results. The client must be connected for the duration of the invocation.

- Point-to-point communication. Invocation is typically targeted for a single object on a particular server.

Mobile clients and large distributed systems motivate the use of asynchronous and anonymous one-to-many distributed computing models. Event-based models address the limitations of the standard client/server paradigm by introducing two roles: consumers and producers. Since event models employ differing technical terms, in this chapter we consider event consumers, listeners, sinks, and event producers, sources and suppliers to be synonymous.

The event model consists of event listeners and event sources. A listener expresses interest in an event supported by an event source, and registers to receive notifications of that event based on a set of parameters. Figure 2.1 presents a general model of the listener-source paradigm, and presents the actual filtering and notification as a black box, which can reside either on the source or on the network. Ideally, the event source does not have knowledge of all the parties that are interested in a particular event.

The event system is a logically centralized component that may be a single server or a number of federated servers. In a distributed system consisting of many servers, there are two approaches for connecting sources and listeners:

- The event service supports subscription of events and routes registration messages to appropriate servers (usually using a minimum spanning tree). One optimization to this approach is to use advertisements, messages that indicate the intention of an event source to offer a certain type of event, to optimise event routing.

- Use some other means of binding the components, for example a lookup service.

In this context, by event listener we mean an external entity that is located on a physically different node on the network. However, events are also a powerful method to enable inter-thread and local communication, and there may be a number of local event listeners that wait for local events.

### 2.2.3 Routing

Event routing requires that store and forward type of event communication is supported within the network on the access nodes (or servers). This calls for intermediate components called event routers. Each event source

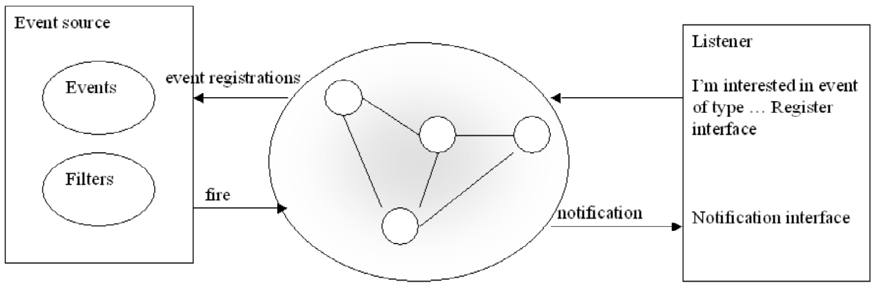


Figure 2.1: General model of the event source and event listener. Event source fires events, and the listener is notified using some mechanism on the network or in the client.

is connected to at least one router. Each router needs to know a suitable subset of other routers in the domain.

In this approach the request, in the worst case, is introduced at every router to get a full coverage of all message listeners. This is not scalable, and the routing needs to be constrained by locality or by hop count. Effective strategies to limit event propagation are zones used in the ECO architecture, the tree topology used in JEDI or the four server configurations addressed in the Siena architecture. Siena broadcasts advertisements throughout the event system, however subscriptions and notifications are routed based on the advertisements, subscription and filters. Multicast works well in closed networks, however in large public networks multicast or broadcast may not be practical. In these environments universally adopted standards such as TCP/IP and HTTP may be better choices for all communication [IBM02a].

### 2.2.4 Content-based Routing

Events are published in a named channel, or in an infrastructure of one or more routers that can use the content of the events in making the forwarding decision. Named channels are also called as topics, and they represent an abstraction of numeric network addressing mechanisms. With content-based addressing clients can change their interests without changing the addressing scheme. With channel-based messaging, new channels need to be added to the address space.

**Content-based** Routing decision is made based on the content, for example strongly typed fields in the event message.

**Subject-based** Routing decision is made based on the subject of the event.

**Channel-based (or topic-based)** Routing decision is made based on the channel on which the event is published. A channel is a discrete communication line with a name.

The producers and consumers must agree on a channel. Content-based and subject-based are more flexible than channel-based messaging, because this agreement is not necessary. Channel-based messaging, however, allows the use of IP multicast groups. The subjects can be allocated to multicast addresses. Channel-based routing can be emulated with content-based systems by limiting to a universally defined subject field.

Content-based event routing has been proposed as one of the requirements for advanced applications, in particular for mobile users [CW01]. Content-based routing takes place above the network level (level 3), and can be based on IP multicast networks, for example. In the content information model, the users subscribe information based on their preferences. The information, when it is available, is then delivered based on these preferences. The subscription paradigm abstracts the publishers of information from the receivers; information is not published to a set of addresses.

Work has been done in using multicast networks to deliver the information to the subscribers [CW01] using multicast addresses. The granularity and flexibility of this approach depends on the size and number of the virtual multicast addresses. As an alternative Carzaniga and Wolf present an application-level information broker with a rich information selection capability.

They define a content-based addressing scheme, by considering the predicates that define subscriptions as the destination addresses. Datagrams are implicitly addressed to a node by their content. The predicate model is a set of boolean functions imposed on the datagram model. Content-based routing is done using an algorithm that uses a forwarding table, which is a map of interfaces to their receiver predicates.

Content-based systems are contrasted with channel-based and subject-based systems, because the selection is done based on the whole content. The other strategies offer only a set of well-defined attributes for selection purposes. The drawback of content-based systems is scalability.

### Filtering

Filtering reduces the number of events sent from the sources to the listeners by matching events against a template. Those events that match the template are forwarded to the listeners. Matching is usually done on single events, but may be also performed on compound events. Filtering improves the scalability of the system. Also, the location of the filtering of events affects the scalability of the framework. Here we face two separate issues: the filtering of simple events and the filtering of compound events. Both kinds of event filtering can be done at several locations:

- At a centralized server (client-server)
- At the listener
- At the event source
- In the infrastructure (event routers)

Source side filtering is more scalable than a centralized server or filtering at the listener. Schemes that use multicasting and listener side filtering place the burden on listeners and the communication infrastructure.

### Quality of Service

Applications based on event-style communication have varying reliability requirements. The event system may support from "at-most-once" semantics to "exactly-once" semantics. In addition, there may be availability, performance, scalability and throughput requirements. The diverse nature of requirements calls for a number of implementations optimized for different sets of requirements.

### Taxonomy

Event models can be grouped into a taxonomy by their properties. As contrasted with the client-server paradigm, event models involve one-to-many communication. Other important aspects for event model classification are [Mei00]:

- Does the model support distributed operation, local operation, or both. In a centralized event model the event sources and listeners are located in the same host, whereas in the distributed model they can be located on different hosts.
- Support for detecting composite events (compound events). Compound events require more complicated filtering/history mechanisms.
- Support for Quality of Service requirements.
- Support for typed events, generic events, or both. Typed events have well-defined structure, for example a set of ordered strings, and generic events do not have an expressive structure (data type any).
- How decoupled the event listeners are from the event sources.
- Is the model subscription-based or advertisement-based.

- Support for channel-based, subject-based or content-based routing.
- Delivery semantics (best-effort, at-most-once..).

Additional aspects are:

- Support for wireless systems and disconnected operation.
- Does the model support event routing, direct notification etc.
- How interests are defined and discovered. Not all models include discovery functionality.

Figure 2.2 presents an example taxonomy based on the event architectures explored in section 2.3.

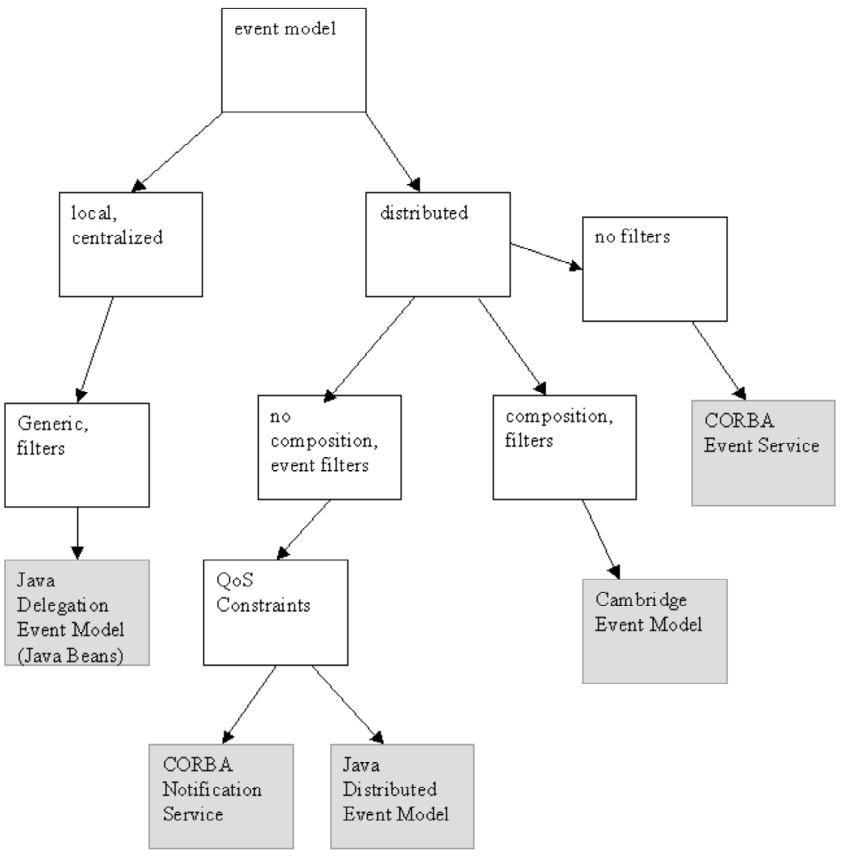


Figure 2.2: Example event model taxonomy.

### 2.2.5 Requirements for Mobile Computing

The mobile environment poses several challenges for detecting and distributing events:

- The network connections are intermittent.
- Bandwidth may vary greatly depending on the connection. This effectively puts constraints on the number of events that can be sent in a certain time interval, how timely they are, and how reliably they can be communicated to the other party.
- The devices may have limited system resources (CPU, memory, storage) and may not have capability to pre-process events but send them as they occur. This motivates an event service located on the fixed network that provides high-level event support for mobile clients.
- The mobile clients may move to a different geographic location or roam a different network. It is preferable that the event service works after a change in connectivity or the service domain.
- The user may wish to share a set of subscriptions between different devices.

Now, we need to consider the following requirements:

- Timely delivery of events, timely being defined in suitable context that is application-specific.
- Reliable delivery of events. Events must be delivered as they are. Events may not become lost.
- Events need to be processed asynchronously.
- Events need to be monitored and notified across domains.
- Event order may not change. If a node sends event A and then event B, their order must be preserved by using time stamping.

In order to support reliable and fault-tolerant event notification, the event sources need to provide reliable persistent storage and buffer events. This is more realizable with fixed network servers, because the mobile clients do not necessarily have persistent storage.

From the device point of view:

- Transmission has a cost both in transmitted bytes and battery life (transmission requires energy).

- How much event history is stored within the device. Distributed event service should be used only for external purposes, not for internal monitoring.
- How to deliver notifications to the device in different networks and protocols. For instance, the bearer may not support push type of communication.

A mediator [BMH+00] (a proxy) can prevent the disconnected mobile user from missing events. In this case, the mediator registers events on behalf of the mobile client and buffers the event notifications. The size of the accumulated set of events may be fairly large. Therefore, the client needs some way to prune the event history and decide what events are crucial for delivery. On the other hand, the client can decide what events are registered, and may deregister unimportant events when the bandwidth is low or costly.

The events may also have a limited temporal existence according to user, system or application requirements. Time-to-live (TTL) timers and hop counters can be used to remove obsolete events.

From the security point of view we have to take into account:

- Encrypting events so that a third party can not capture them while in transit.
- Securing the event service and the event bus.
- Access control for registering event listeners.

## 2.3 Event Systems

This section presents event model implementations. We start from the standard centralized event model in Java, and continue with the Distributed Event Model in Java. We present the Java Messaging Service in subsection 2.3.3, subsection 2.3.4 presents the CORBA Event Service and subsection 2.3.5 the Notification Service. In subsection 2.3.6 we examine the CORBA Management of Event Domains; subsection 2.3.7 presents the Cambridge Event Architecture and subsection 2.3.8 the Siena architecture. In subsection 2.3.9 we overview Elvin, and subsection 2.3.10 presents a number of other event systems.

### 2.3.1 Java Delegation Event Model

The Java Delegation Event Model was introduced in the Java 1.1 Abstract Windowing Toolkit (AWT) and serves as the standard event processing method in Java. The model is also used in the Java Beans architecture and supported in the PersonalJava and EmbeddedJava environments.

In essence, the model is centralized and a listener can register with an event source to receive events. Event source is typically a GUI element and fires events of certain types, which are propagated to the listeners. The event delivery is synchronous, so the event source actually executes code in the listener's event handler. No guarantees are made on the delivery order of the events [Mei00].

The event source and event listener are not anonymous, however the model provides an abstraction called an adapter, which acts as a mediator between these two actors. The adapter decouples the source from the listener and supports the definition of additional behaviour in the event processing. The adapter may implement filters, queuing and QoS controlling.

### 2.3.2 Java Distributed Event Model

The Distributed Event Model of Java is based on Java Remote Method Invocation (RMI) that enables the invocation of methods in remote objects. This model is used in Sun's Jini architecture. The architecture of the Distributed Event Model is similar to the architecture of the Delegation Model with some differences.

The model is based on the Remote Event Listener, which is an event consumer that registers to receive certain types of events in other objects. The specification provides an example of an interest registration interface, but does not specify such. The Remote Event is the event object that is returned from an event source (generator) to a remote listener. Remote events contain information about the occurred event, a reference to the event generator, a handback object that was supplied by the listener and a unique sequence number to distinguish the event globally. The model supports temporal event registrations with the notion of a lease (Distributed Leasing Specification). The event generators inform the listeners by calling the listeners' notify method. The specification supports Distributed Event Adaptors that may be used to implement various QoS policies and filtering.

The handback object is the only attribute of the Remote Event that may grow to unbounded size. It is a serialized object that the caller provides to the event source; the programmer may set the field to null. Since the handback object carries both state and behaviour it can be used in many ways, for example to implement an event filter at a more powerful host than the event source. A mediator component can register to receive events, and gives a filter object to the source. Upon event notification, the filter is handed back and the mediator can use it to filter event before handing to the original event listener.

The specification supports recovery from listener failures by the notion of leasing. Lease imposes a timeout for event registrations. This is used to ease the implementation of distributed garbage collection. Since this model relies on RMI, it is inherently synchronous. Each notification contains a sequence number that is guaranteed to be strictly increasing.

### 2.3.3 Java Message Service (JMS)

JMS (Java Messaging Service) [Sun01] defines a generic and standard API for the implementation of message-oriented middleware. The JMS API is an integral part of the Java Enterprise Edition version 1.3. The J2EE supports the message driven bean, a new kind of bean that enables the consumption of messages. However, JMS is an interface and does not provide any concrete implementation of a messaging engine. The fact that JMS does not define the messaging engine or the message transport gives rise to many possible implementations and ways to configure JMS. JMS supports a point-to-point (queues) model and a publisher/subscriber (topics) model. In the point-to-point model only one receiver is selected to receive a message, and in the publisher/subscriber model many can receive the same message.

The JMS API can ensure that a message is delivered only once. At lower levels of reliability an application may miss messages or receive duplicate messages. A standalone JMS provider (implementation) has to support either point-to-point or the publish/subscribe approach or both. Normally, JMS queues and topics are maintained and created by the administration rather than application programs. Therefore the destinations are seen as long lasting. The JMS API also allows creating temporary destinations that last only for the duration of the connection.

The point-to-point communication model consists of receivers, senders and message queues. Each message queue is addressed to a particular queue, and receivers extract messages from the queues. Each message has only one consumer and the client acknowledges the successful delivery of a message to the component that manages the queue. In this model there are no timing dependencies between a sender and a receiver, it is enough that the queue exists.

In addition, the JMS API allows the grouping of outgoing messages and incoming messages and their acknowledgements to transactions. If a transaction fails, it can be rolled back.

In the publish/subscribe model the clients address messages to a topic. Publishers and subscribers are anonymous, and messaging is usually one to many. This model has a timing dependency between consumers and producers. Consumers receive messages after their subscription has been processed. Moreover, the consumer must be active in order to receive messages. The JMS API provides an improvement on this timing dependency by allowing clients to create durable subscriptions. Durable subscriptions introduce the buffering capability of the point-to-point model to the publish/subscribe model. Durable subscriptions can accept messages send for clients that are not active at the time. A durable subscription can have only one active subscriber at a time.

Messages are delivered to clients either synchronously or asynchronously. Synchronous messages are delivered using the receive method, which blocks until a message arrives or a timeout occurs. In order to receive asynchronous messages, the client creates a message listener, which is similar to an event listener. When a message arrives the JMS provider calls the listener's onMessage method to deliver the message.

JMS clients use JNDI to look up configured JMS objects. JMS administrators configure these components using provider (implementation) specific facilities. There are two types of administered objects in JMS: ConnectionFactories, which are used by clients to connect with a provider, and Destinations, which are used by clients to specify the destination of messages.

JMS messages consist of a header with a set of header fields, properties that are optional header fields (application-specific, standard properties, provider-specific properties), and a body that can be of several types. Message selection is supported by filtering the message header against the given criteria using an SQL grammar. A JMS message selector allows clients to define the messages they are interested in. Headers and properties need to match the client specification in order to be delivered to that client. Message selectors cannot reference values embedded in the message body. For example: "JMSType='stock' AND company='abc' AND stockvalue > 100"

JMS supports five different messages types: Map, Object, Stream, Text, and Bytes. MapMessage is a set of name/value pairs, where names are strings and values are primitive Java types. ObjectMessage is a message containing a serializable Java object. StreamMessage is a stream of sequential Java primitive values. TextMessage represents an instance using the java.util.string class and can be used to send and receive XML messages. BytesMessage is a stream of bytes.

Typically a JMS client creates a Connection, one or more Sessions and a number of MessageConsumers and MessageProducers. Connections are created in the stopped mode. After a connection is started (start() method) messages start arriving to the consumers associated with that connection. A MessageProducer can send messages while a Connection is stopped. A Session is a single-threaded context for consuming and producing messages. Sessions act as factories for creating MessageProduces, MessageConsumers and temporary destinations. JMS defines that messages sent by a session to a destination must be received in the order in which they were sent.

Messages are acknowledged automatically in the transactional mode (supported by the Java Transaction API), however if a session is not transacted there are three possible options for doing acknowledgement: lazy acknowledgment that tolerates duplicate messages, automatic acknowledgement, and client-side acknowledgement. In persistent mode delivery is once-and-only-once, and in non-persistent mode the semantics are at-most-once.

JMS messaging proceeds in the following fashion:

1. Client obtains Connection from ConnectionFactory
2. Client uses Connection to create a Session object
3. Session is used to create MessageProducer and MessageConsumer objects, which are based on Destinations.
4. MessageProducers are used to produce messages that are delivered to destinations.
5. MessageConsumers are used to either poll or asynchronously consume (using MessageListeners) messages from producers.

The JMS API (1.0.2b) does not address load balancing, fault tolerance, error notification, administration, or security. JMS implementations are available from many vendors, such as IBM (it is supported in MQSeries), Sun Microsystems (J2EE), The ExoLab Group (OpenJMS), SoftWired (iBus/-/Mobile), Oracle (8i and later) etc.

### JMS and CORBA Interoperability

The communication models of JMS and CORBA are similar, however integration is necessary in the areas of message conversion, filtering, and the incorporation of point-to-point mode, which uses queues (CORBA uses publish-subscribe). The Notification Service supports structured events defined in IDL, and JMS supports the five different message formats.

OMG is working on a Notification Service / JMS Interworking document [OMG02a] and currently the initial submission deadline has passed with a submission from Alcatel, Fujitsu, IONA, and PrismTech. The RFP deals with mappings between message types, reconciliation between different QoS properties, the ability to maintain transactional message contexts across the services, and implementations which facilitate end-to-end messaging between the services.

The specification defines a bridge that manages and interconnects an event channel with a JMS destination. The principles behind the Bridge IDL definitions were to provide backward compatibility with the programming models of NS and JMS. The Bridge is a stateful entity that mediates messages between the two systems. Structured events are used to improve performance. The Bridge is also used to automate the connection setups between channels and destinations. A BridgeFactory object supplies Bridge objects depending on the parameters: the channel, destination, type of communication (push/pull), and message type (sequence, single). Since JMS does not support pull at the source side, this is not supported.

In the implementation of PrismTech's OpenFusion [Pri01], the JMS event producer is extended by a client-side library that transforms JMS messages to CORBA Notification Service structure events. JMS consumers may use push and pull, but the consumers of the Notification Service may only use one of these two approaches.

JMS only allows clients to specify filters on the message properties. To keep the information filterable, this data needs to be included in the filterable body of a structured event. The JMS message interface supports three attributes that are also supported in the Notification Service:

1. DeliveryMode (persistent, non-persistent which maps to best effort in CORBA NS)
2. Expiration (expiration in milliseconds, set to QoS in the variable Timeout)
3. Priority (Mapped to notification Priority QoS in the variable header)
4. Other user-defined name-value pairs are converted to IDL using the standard primitive mapping.

Since Notification Service uses the Extended Trader Constraint Language and JMS uses the where clause of SQL92, the Notification Service needs to be extended to support SQL92.

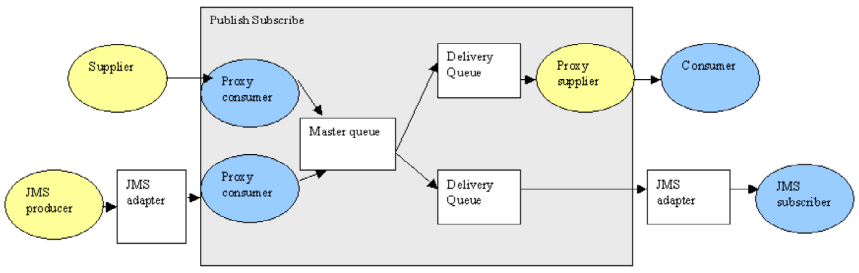


Figure 2.3: The OpenFusion Notification Service with JMS publish-subscribe interoperability.

### Wireless JMS

The iBus//Mobile software from SoftWired consists of a server-side gateway for mobile clients and a JMS compatible messaging server (iBus//MessageServer). The gateway enables communication between a wide variety

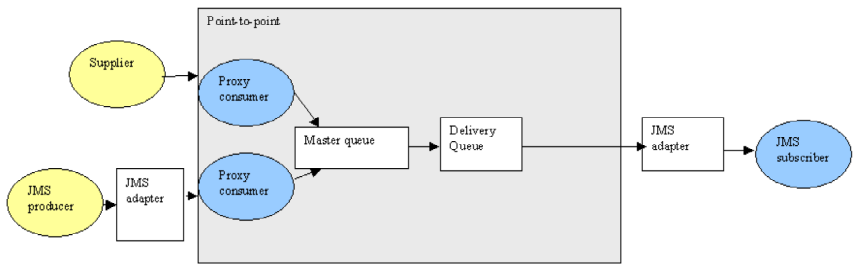


Figure 2.4: The OpenFusion Notification Service with JMS point-to-point interoperability [Pri01]

of devices running different operating systems, such as PalmOS, Symbian, and PocketPC. The gateway supports communication over SMS, WAP, TCP, UDP, and GPRS. The system supports corresponding Java virtual machines, J2ME (CLDC and CDC), PersonalJava, and J2SE [$R^+01a$].

All the communication between clients and the gateway is transmitted in binary form. From the JMS provider's viewpoint the gateway is a regular JMS client and from the client's viewpoint the gateway is a communications hub and a wrapper for different transport and representation formats. In the case of SMS the gateway accepts the incoming messages and a component within the service domain can respond with SMS.

The client side library takes a minimum of 70k and at runtime the CLDC version takes a minimum of 50k of Java heap (as a comparison a 8MB Palm has a 150k Java heap). The iBus system supports security in the form of access control, certificates, and symmetric/antisymmetric keys. Cryptographic functions are supported from third-party libraries. If the bearer does not support push-type connections, one connection is used for sending client data to the server and another connection is used for communication from the gateway to the client. Each HTTP request goes over the first connection: send data to the servlet, and return. The second connection is open and blocks until there is traffic; after receiving messages the connection is immediately re-established. The underlying library hides the differences between the protocols.

### 2.3.4 The CORBA Event Service

The CORBA Event Service specification (current version 1.1) defines a communication model that allows an object to accept registrations, and send events to a number of receiver objects [Sie99]. The Event Service supplements the standard CORBA operational call client-server communication model and is part of the CORBAServices that provide system level services for object-based systems. In the client-server model illustrated in Figure 2.5, the client makes a synchronous IDL operation on a specified object at the server. The event communication is unidirectional (CORBA oneway operations) [OMG01a].

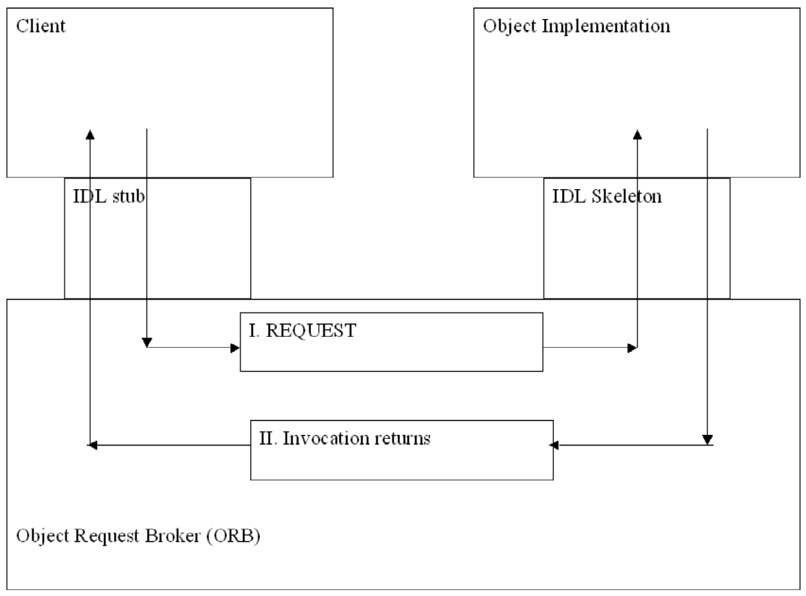


Figure 2.5: The standard CORBA client-server model of invoking operations from client to the target object.

The Event Service extends the basic call model by providing support for a communication model, where client applications can send messages to arbitrary objects in other applications. The Event Service addresses the limitations of the synchronous and asynchronous invocation in CORBA.

The specification defines the concept of events in CORBA: an event is created by the event supplier and is transferred to all relevant event consumers. The set of suppliers is decoupled from the set of consumers, and the supplier has no knowledge of the number or identity of the consumers. The consumers have no knowledge of which supplier generated the event.

The Event Service defines a new element, the event channel, which asynchronously transfers events between suppliers and consumers. Suppliers and consumers connect to the event channel using the interfaces supported by the channel. An event is a successful completion of a sequence of operation calls made on objects: consumers, suppliers, and the event channel.

The event channel performs the following functions:

- It allows consumers to register interest in events, and stores the registration information.
- It accepts events generated by suppliers.
- It forwards events from suppliers to registered consumers.

The Event Service is defined to operate above the ORB architecture: the suppliers, the consumers, and the event channel may be implemented as ORB applications and events are defined using standard IDL invocations.

### Push and Pull

The CORBA Event Service provides two models for initiating the transfer of events between suppliers and consumers. The first model is the push model, in which suppliers send events to consumers (Figure 2.6). In this case, the suppliers are active, and the consumers are passive. Moreover, the event channel actively delivers events to the consumers. In the second model, the pull model (Figure 2.7), the consumers request events from the suppliers. Now, the consumer actively waits for pull requests to arrive. Upon the arrival of a pull request, the event is generated and sent to the pulling consumer. CORBA supports both blocking and non-blocking pull.

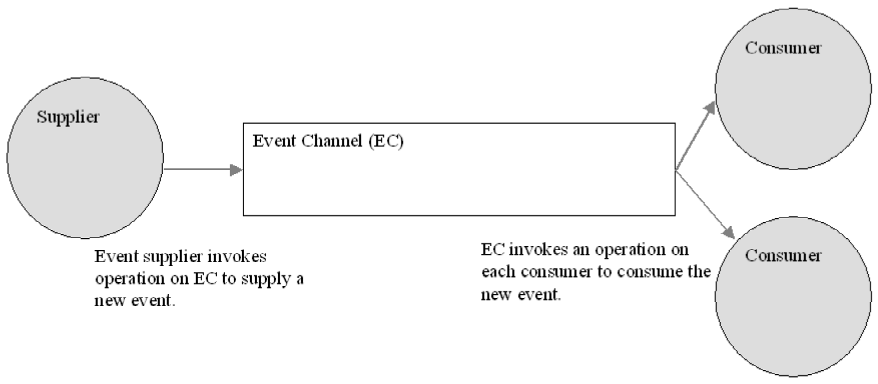

Supplier
Event Channel (EC)
Consumer
Consumer
Event supplier invokes operation on EC to supply a new event.
EC invokes an operation on each consumer to consume the new event.

Figure 2.6: Example of an event propagation implementation.

### The Hybrid Model

It is also possible to mix the push and pull models in one application, because the event channel decouples the consumers and the suppliers from each other. It is possible to connect suppliers using the push model and consumers using the pull model. In the hybrid model, the event channel does not take an active role in delivering the event to the consumers.

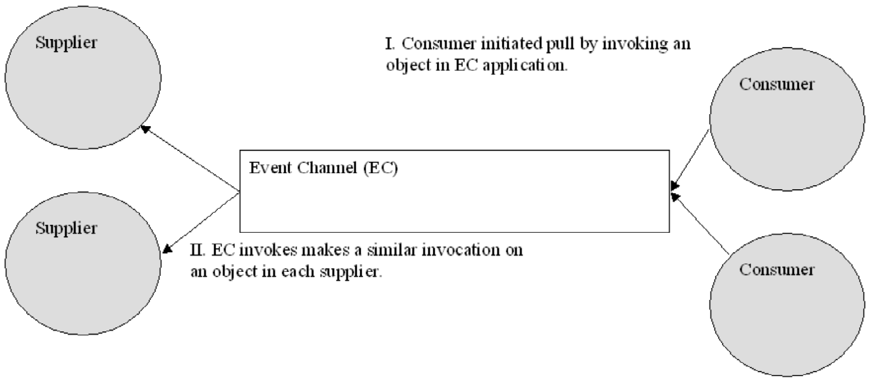

Supplier
Supplier
I. Consumer initiated pull by invoking an object in EC application.
Consumer
Consumer
Event Channel (EC)
II. EC invokes makes a similar invocation on an object in each supplier.

Figure 2.7: Pull Model and the Event Channel.

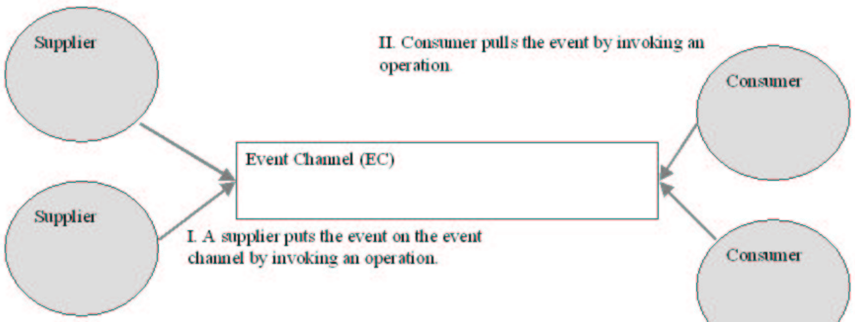

Supplier
Supplier
II. Consumer pulls the event by invoking an operation.
Consumer
Consumer
Event Channel (EC)
I. A supplier puts the event on the event channel by invoking an operation.

Figure 2.8: The hybrid model mixing Push and Pull models.

### Connecting Suppliers and Consumers

The Event Service specification does not include a mechanism for locating or discovering consumers or suppliers, however it provides the administrative operations for connecting the suppliers and the consumers. Each new event consumer added to the event channel returns a proxy supplier. The proxy supplier follows the supplier interface and adds a new method for connecting a consumer to the proxy supplier. Each new event supplier added to the event channel, returns a proxy consumer. The proxy consumer has a new method for connecting to the proxy supplier.

A supplier is registered by taking a proxy consumer from the event channel and connecting it with the supplier. Similarly, an event receiving application takes a proxy supplier from the event channel and connects to it by providing a consumer. Each admin object is a factory that creates the proxy interface that is used in connecting the clients and the event sources. Consumer admins create proxy suppliers and supplier admins create proxy consumers.

### Typed and Untyped Event Communication

The data about an event can be passed as invocation parameters or return values. Events are not objects, because the CORBA object model does not support passing objects by value (2.3 supports valuetypes). Event data is application specific and can be either untyped or typed.

In untyped communication the event is propagated by invoking a series of generic push and pull operations. The push operation takes a single parameter of the type any, which allows any IDL defined data type to be propagated, and stores the event data. The pull operation has no parameters and transfers event data in its return value, which supports the type any. In untyped communication both the supplier and the consumer applications need to agree on the data format of the event.

In typed event communication events are propagated through an application specific interface created by the programmer in IDL. The programmer defines the interface for event propagation that is used by consumers and suppliers. Parameters can be of any suitable data type supported by the IDL language.

To setup typed push-style communication, the consumers and suppliers exchange object references (TypedPushConsumer and PushSupplier). The supplier invokes a method to get a reference that supports the typed consumer interface I. The particular reference is associated with the TypedPushConsumer interface and needs to be agreed on by both the consumer and the supplier. The supplier uses this reference to invoke operations on the consumer.

In the typed pull model consumers request event information using some mutually agreed interface. The parties exchange the PullConsumer and TypedPullSupplier interfaces, and an object reference supporting the typed interface is obtained. Once the reference is obtained, the consumer can invoke operations on the supplier.

### Discussion

The CORBA Event Service supports different implementations of the Event Channel, and this allows a wide range of approaches for implementing Quality of Service and delivery issues. Moreover, the event consumer and supplier interfaces support disconnection.

The CORBA Event Service addresses some of the problems of the standard CORBA synchronous method invocations by decoupling the interfaces and providing a mediator for asynchronous communication between consumers and suppliers. The supplier does not have to wait for the event to be delivered to the consumer. Moreover, the event channel hides the number and identity of the consumers from suppliers using the proxy objects (transparent group communication). The supplier sends events to its proxy consumer, and the consumer receives events from its proxy supplier.

However, the specification does not address several important issues, such as Quality of Service support. Applications may have requirements for event notification in terms of reliability, ordering, priority and timeliness. Furthermore, the specification does not provide a system for event filtering. Event filtering needs to be implemented using a proprietary system within the event channel by adding a mechanism for selective event delivery. Event channels can be composed, because they use the same consumer/-supplier interfaces. An event channel can push an event to another event channel. Typed event channels can be used to filter events based on event type [Bar01] [OMG01a].

In addition, the specification does not address compound events, but suggests that complex events may be handled by creating a notification tree and checking event predicates at each node of the tree. The drawback of the tree is that the number of hops needed to deliver an event increases. This motivates the use of a centralized filtering service.

The use of proprietary event service implementations restricts the interoperability of applications. Applications that use one proprietary event service implementation may not interoperate with another application that is based on a different event service implementation.

### 2.3.5 CORBA Notification Service

The CORBA Notification Service (current version 1.0) [OMG01b] extends the functionality and interfaces of the Event Service to support better interoperability [Bar01]. One of the most significant additions to the Notification Service is event filtering. Filters allow consumers to receive particular events that match certain constraint expressions. Filtering reduces the number of events sent to the consumers and improves the scalability of the event handling system, limiting the scalability of the mechanism.

Figure 2.9 presents the components in the CORBA Notification Service, which derive from the Event Service discussed in the previous section. The event channel has been extended to support a number of admin objects. The Notification Service allows the definition of filters at the proxies. Moreover, each admin object is seen as the manager of the set of proxies it has created. Admin objects may be associated with QoS properties and filter objects. The QoS properties and filter objects of the admin object are transferred to each proxy it creates, however the QoS properties may be changed on a per proxy basis.

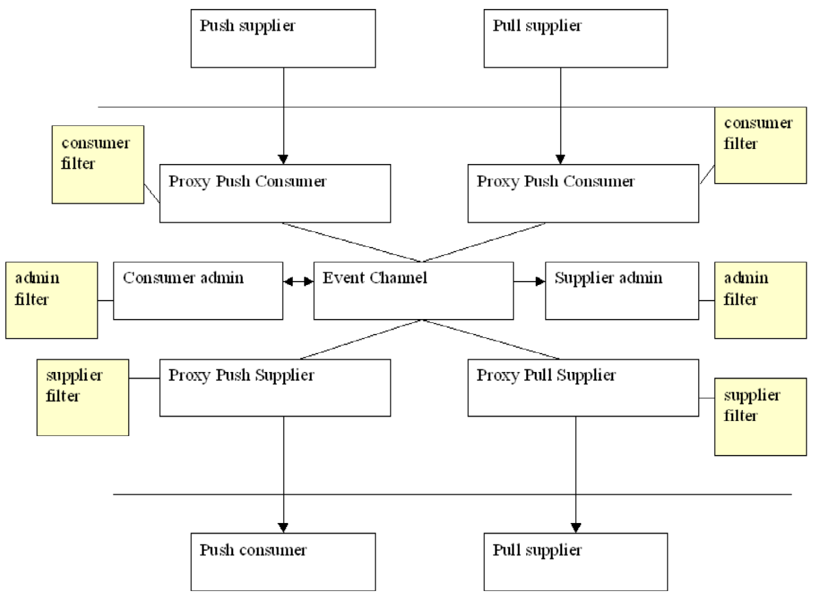


Figure 2.9: Components in the CORBA Notification Service [GCSO01].

### Filters

Filters are CORBA objects that support the addition, modification, and removal of constraints. Constraints are used to match event message values and refer to variables that are part of the event notification message. Constraints are either event types or written in a constraint language. Variable names can refer to all parts of the current notification. The current notification is expressed with the dollar sign '$'.

A sample notification constraint:

```
$.type_name == StockAlert
$.market_name == 'NASDAQ'
$.ticker == 'Company'
$.price > '100' or $.price < 80
```

The default constraint grammar is Extended TCL (Trader Constraint Language specified by the Trading Service). The Event Notification specification adds the notion of mapping filter objects. Each proxy supplier may have an association with a mapping filter object, which affects the priority of the events it receives and the lifetime property of the events it receives.

### Quality of Service (QoS)

The Notification Service defines standard interfaces that allow the control of characteristics over the delivery of the notification. Service characteristics at different levels in the protocol stack are represented using name/value pairs. QoS properties, tuples of form <String, Any>, can be used with an event channel, admin objects, proxy suppliers, proxy consumers, and message instances.

Characteristics include:

- Discard policy that determines which notifications are discarded when resource limits apply (queues are full).
- Earliest delivery time.
- Expiration time, which indicates the time range when the event is valid.
- Maximum number of notifications that can be queued for a single consumer. This effectively places an upper bound that lessens the load presented by misbehaving consumers.
- Order policy, which specifies the order in which notifications are buffered for delivery.
- Priority of events.
- Reliability of event delivery
- Both event reliability and connection reliability. If fault-tolerance properties are specified, the Notification Service reconnects to the set of clients and delivers all non-expired events to consumers after a crash or disconnection. At the message level: Best effort, persistent.

Furthermore, the event channel supports the following QoS properties:

- MaxQueueLength, which specifies the maximum number of events that can be queued.
- MaxConsumers, which specifies the maximum number of consumers that can be connected to the channel.
- MaxSuppliers, which specifies the maximum number of suppliers that can be connected to the channel.

### Structured Events

The Notification Service defines a standard data structure for the events. The structured event illustrated in Figure 2.10 is a strongly typed event message that consists of a header and a body. The header contains two sections:

- the first stores fixed information, such as domain_name, event_name, and type_name.
- The second section stores the variables and optional information about the event. This is a sequence of properties to hold QoS information related to the notification.

The body of the structured event stores the actual event data, and it is also divided into two sections:

- the filterable data, which is a sequence of properties. This part contains the fields that the consumers use to base filtering decisions on.
- the payload data.

The header and body are structured into two parts mainly because of performance reasons. When filterable data has its separate compartment, it is not necessary to touch the payload data upon filtering. Moreover, the notification could be contained within the optional header fields leaving the body empty. This would be even more streamlined.

### Discussion

Since the Event Channel is a CORBA object, it limits the number and complexity of any given event channel. Therefore, it becomes important to create, manage, and specify federations of event channels. Each event channel has a master queue and a number of consumer queues. Each queue has some maximum capacity, which may be enforced using QoS policies supported by the specification. One way to relieve the bottleneck of the centralized event channel is to distribute these queues as CORBA objects, however this kind of solution is still centralized. Since NS supports the federation of channels by connecting the supplier and consumer proxies, the system supports scalability.

Channel federation can be used to:

- Improve performance by distributing consumers on several event channels. Since an event channel is a CORBA object, it may become a bottleneck if the number of consumers (or producers) becomes large. Event channels may also be used to enhance local delivery by assigning each event channel only local subscribers. In this case we have only one network invocation, and a number of local invocations.

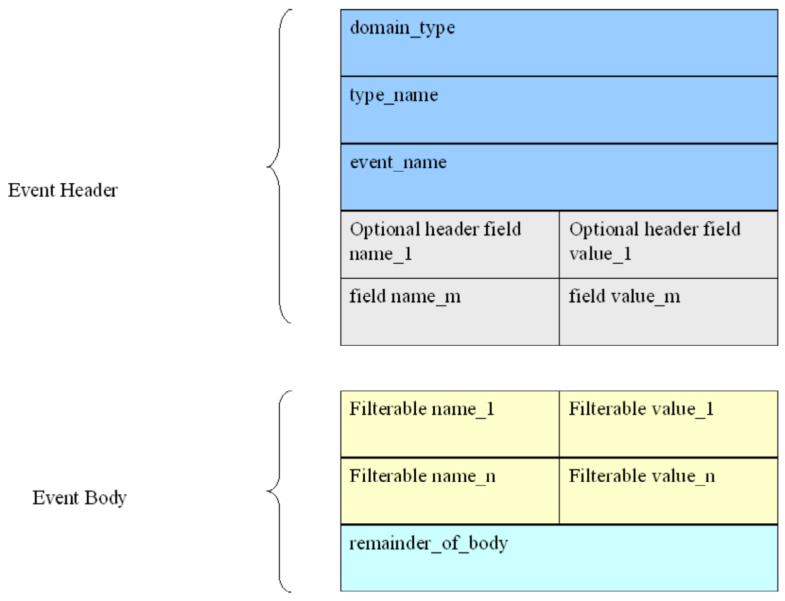


Figure 2.10: The structured event: Event header and event body.

- Improve reliability by having multiple event channels for the same information. If one event channel fails, it does not necessarily prevent consumers from receiving the notifications.
- Improve flexibility by grouping consumers and producers into logical units (event channels).

Currently the initial submission deadline for an RFP (Request For Proposals) for Realtime Notification has passed and the deadline for revised submissions is June 3, 2002. This specification would extend the notification service with predictable (bounded) notification behaviour in order to support realtime and safety-critical systems.

### 2.3.6 CORBA Management of Event Domains

CORBA Event Service and Notification Service do not specify an event discovery service or a mechanism how to federate event channels. Moreover, the procedure for connecting event channels is complex. The OMG Telecommunications Domain Task Force addresses these issue in the CORBA Management of Event Domains Specification [OMG02b], which specifies an architecture and interfaces for managing event domains. An event domain is a set of one or more event channels grouped together for management

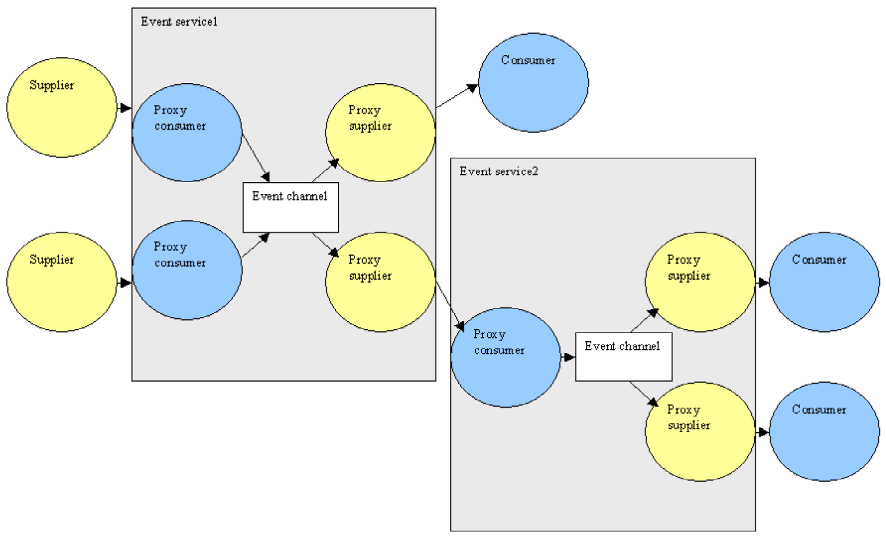


Figure 2.11: CORBA Notification Service channel federation.

and improved scalability. The specification defines two generic domain interfaces for managing generic, typed, and untyped channels. Moreover, a specialized domain for both channels and logs defined by the OMG Telecom Log Service specification.

The specification addresses [OMG02b]:

- Connection management of clients to the domain.
- Topology management
- Sharing the subscription and advertisement information in an event domain, even when connections between event channels change at runtime.
- Event forwarding within a channel topology.
- Connections between event channels.

It supports the creation of channel topologies of arbitrary complexity, allowing cycles and diamond shapes in the graph of interconnected channels. However, if events may reach a point in the graph by more than one route duplicate events need to be detected and removed. Moreover, if no timeouts are specified, events in a cycle will propagate infinitely. Therefore, the specification defines mechanisms that are used to detect cycles or diamonds in the network topology. Graph topology enforcement is done at channel connection time, and illegal connections are refused by the domain management.

Event suppliers inform the proxy consumers of event type changes using the offer_change callback. The channel is responsible for sharing this information with the consumers by executing offer_change on them. The consumer may be another channel and thus the change may propagate throughout the channel topology. Subscription changes work similarly, and the channel is responsible for invoking the subscription_change operation on all the suppliers.

Event suppliers attached to the channel can obtain the types of subscriptions of event channels anywhere downstream by invoking obtain_-subscription_types on the proxy consumers. Similarly event consumers can obtain the event types offered by suppliers on any event channel downstream by invoking obtain_offered_types on its supplier channels.

### 2.3.7 The Cambridge Event Architecture

The Cambridge Event Architecture (CEA) uses the publish-register-notify paradigm [BMH+00], in which the object publishes its interface, for example specified in IDL (Interface Definition Language, which is different from the IDL in CORBA). This interface includes the events it is capable of notifying. A client invokes the object synchronously and can register for events by indicating parameters or wildcards. The template system provides a rudimentary filtering by matching parameters one by one. The object accepts registrations and notifies the clients that match the registration template. The notification is performed when the event firing conditions and access restrictions are satisfied (Figure 2.12). The paradigm supports direct source-to-client event notification.

In CEA an object, if asked, publishes the events it is capable of notifying in IDL. The object has a register method in its interface that has parameters for the type of event and wildcards. Event occurrences are objects of a specific type, and the set of types defines the level of event detection and notification granularity. CEA enforces access control upon registration, and authentication is based on a parameter value.

CEA supports the definition of intermediate services, which are called event mediators in the architecture. Event mediators act as middlemen between primitive event sources and the event clients, and provide the facilities for detecting more complex events. Moreover, if the event source cannot afford the overhead to support template matching, it can send all its events to the mediator. The mediator then matches the template on behalf of the source.

The mediator is capable of providing equivalent functionality to the CORBA event service. The CORBA event service registers interests in all notifiable events with event sources and supports both a synchronous pull

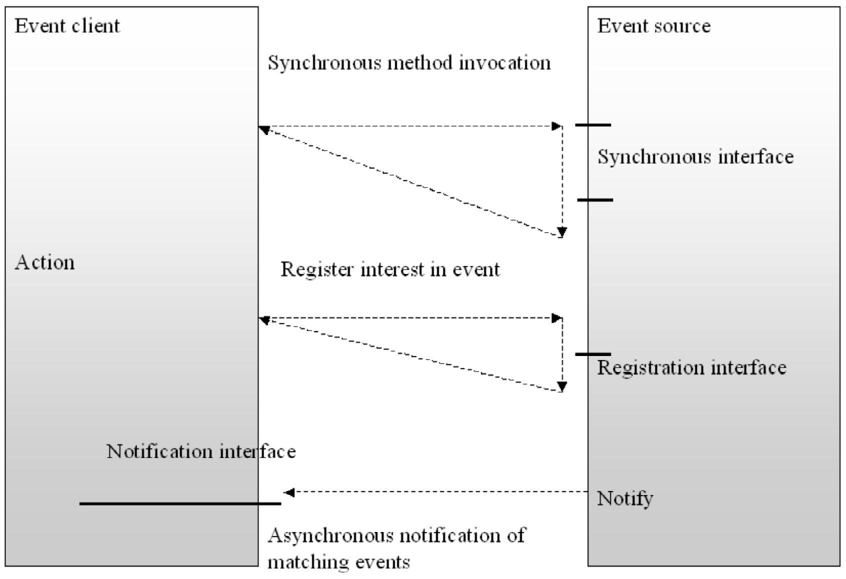


Figure 2.12: A publish-register-notify event architecture [BMH+00].

interface and an asynchronous push interface. Composite events can be detected by giving the mediators the capability to filter simple events of different types across different sources.

The composite event detection functionality supported in CEA is a feature that is not present in many event systems. The event composition is supported by the combination of event templates. Composite events are detected by monitors, which are busy until the event is detected and notified. A composite event specification language may be used to design a monitor that detects complex templates. The system has been demonstrated by implementing an active badge system that monitors badges within a building.

### 2.3.8 Scalable Internet Event Notification Architecture

Siena (Scalable Internet Event Notification Service) is an Internet-scale event notification service developed at the University of Colorado. Siena balances expressiveness with scalability and explores content-based routing in a wide-area network. The basic publish-subscribe mechanism is extended with advertisements that are used to optimize the routing of subscriptions [CRW99].

Several network topologies are supported in the architecture, including hierarchical, acyclic peer-to-peer, and general peer-to-peer topologies.

Servers only know about their neighbours, which minimizes routing table management overhead. Servers employ a server-server protocol to communicate with their peers and a client-server protocol to communicate with the clients that subscribe to notifications. It is also possible to create hybrid network topologies.

Siena is similar to IP-multicast, however the two mechanisms differ in the way they support the groups of subscribers. IP groups are not very expressive. They partition the IP datagram address-space and each datagram belongs at most to one group. Clearly, this creates problems if an event is to be delivered that spans several groups of subscribers.

Four different server topologies have been identified in Siena:

- Centralized
- Hierarchical (Figure 2.13)
- Acyclic peer-to-peer (Figure 2.14)
- Generic peer-to-peer (Figure 2.15)

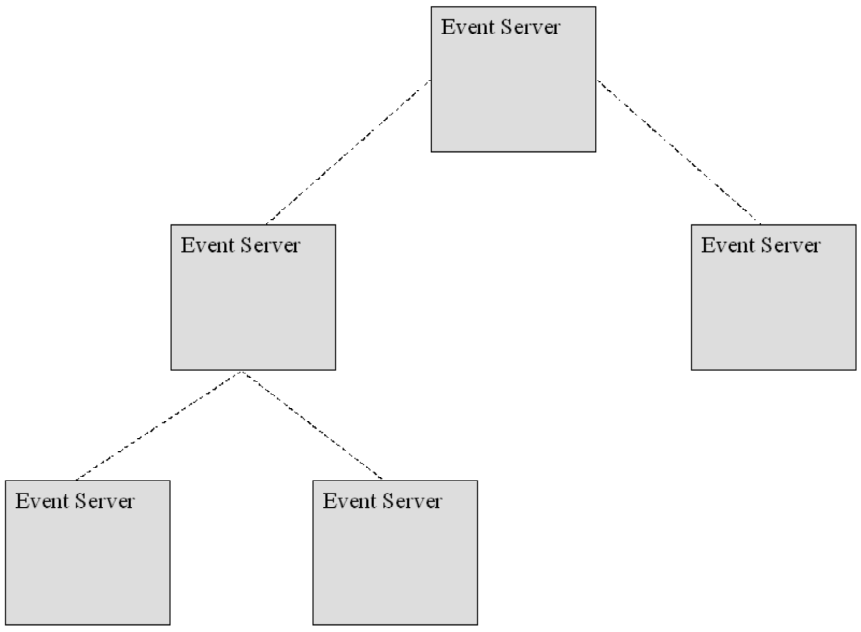


Figure 2.13: Hierarchical configuration. Dotted lines represent client-server protocol.

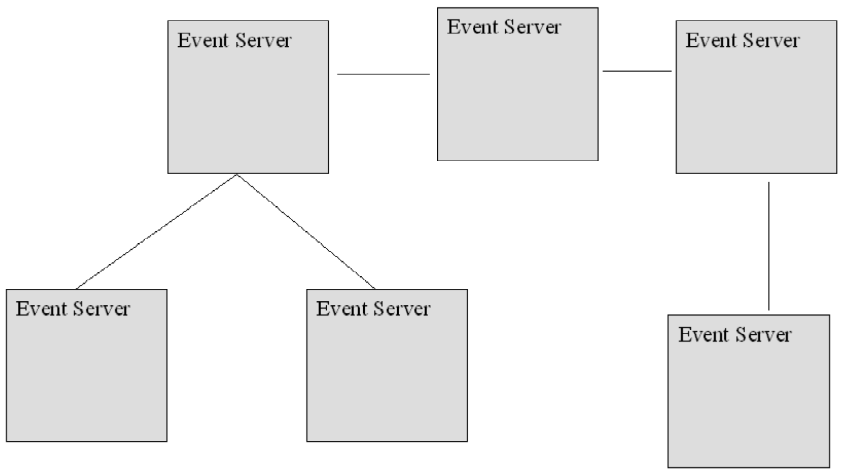


Figure 2.14: Acyclic peer-to-peer configurations. Solid lines indicate server-server protocol.

### Naming and Filtering

Siena is implemented with a flat event namespace, that is, event names have no structural correlation with each other. An event consists of a set of attribute-value pairs. Each attribute has a name and a value. Siena supports the following types: null, string, long, integer, double, and boolean.

A filter has the form of attribute name, constraint operator, and constraint. Siena does not support wildcards in the attribute name so the attribute names must match exactly to the names in the published event. A filter may include several filtering clauses, which are ANDed together. Thus every filtering clause or component must return true in order for the filter to pass the event. Siena supports the following operators: equal, less than, greater than, greater than or equal to, less than or equal to, string prefix, string suffix, always matches, not equal, and substring.

An example event:

```
string stock "abc"
int value     2.53
```

An example filter:

```
string stock = "cde"
int value > 1.0
int value < 1.5
```

Siena supports patterns, which are based on the event attribute values and event combinations. A pattern is a sequence of filters that is matched

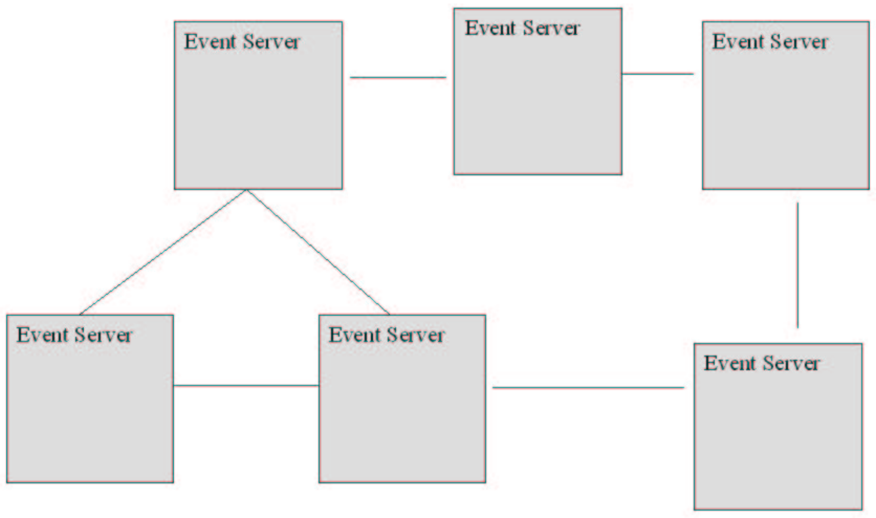


Figure 2.15: Generic peer-to-peer configuration. Solid lines indicate server-server protocol.

to a temporally ordered sequence of notifications. Network latencies may cause some events to arrive in the wrong order, and these are ignored by the Siena solution.

### Routing

In Siena, each event consists of a set of attribute-value pairs that are matched with filters. Each server on the event system routes events to other servers based on the subscription-information, advertisement information, and filters. Each subscriber may specify a filter to constrain the subscription. In the same fashion, each advertisement may also include a filter. Siena evaluates the filters and follows a policy, where events are replicated downstream and filtered upstream. This means that events are replicated to the clients at the last possible moment, thus reducing bandwidth necessary to transmit the events. Upstream filtering means that events are filtered as close to the sources as possible in order to reduce the number of uninteresting events transmitted over the network. Simple filter syntax allows the decomposition of a complex filter into several more general filters, which can be evaluated upstream. A filter is only applied if it is less general than the one used in upstream.

The same principle of upstream filtering applies also to event patterns. Patterns are decomposed (factored) into elementary filters that are delegated to other servers. In the delegation process a server tries to assemble subpatterns that are delegable to other servers.

Siena uses covering relations to determine when a filter covers a notification, a subscription covers a notification, an advertisement covers a notification and an advertisement covers a subscription. For example, subscription S1 covers S2 if it evaluates to true in every instance where S2 is true. Servers propagate the most generic subscription that covers a given set of subscriptions. This minimizes the downstream data structures, however, the complex computation cost is paid closer to the subscriber, because the subscriptions need to be matched and evaluated. The results of Siena indicate that the covering relations exhibit a complexity that is quite reasonable for a scalable service.

The Siena system supports two different notification semantics: subscription-based semantics and advertisement-based semantics. In subscription-based semantics subscriptions are introduced at every node of the event service and a notification is routed if it covers a subscription. In advertisement-based routing servers use the information provided by event producers to route incoming subscriptions. A subscription is only forwarded if it covers the advertisement.

### Forwarding algorithm

The forwarding algorithm that was developed in conjunction with the Siena project consists of a forwarding table and a set of processing functions. Conceptually the forwarding table is a mapping between predicates and interfaces to neighboring nodes. Each predicate is a disjunction of filters, where each filter is a conjunction of elementary conditions. Therefore each filter must return true in order for a predicate to map to an interface. Each filter may map to several interfaces [CDW01].

The forwarding algorithm iterates over the event attributes. It searches for a partial match from the set of filters, where a constraint belonging to a filter is matched by the given attribute. If the filter (with the partial match) is not yet associated with an interface, the algorithm increases a counter to keep track of matched constraints for the given filter. In addition, if the counter size is equal to the number of constraints in the filter, the filter is said to match. After processing one filter the algorithm checks if all filters are matched. The algorithm stops if either all attributes or all filters are processed.

The number of interfaces thus imposes an upper bound on the processing along with the number of attributes and filters. The pseudocode presented in [CDW01] uses a one-to-one mapping of filters and interfaces, which leads to incorrect behavior if one filter is associated with several interfaces. The forwarding algorithm is optimized using binary trees and lookup indexes for attributes used in the filters.

The performance and scalability of the forwarding algorithm were demonstrated by running experiments with 1000 messages and various numbers of filters and other parameters. It was found that the algorithm has good absolute performance and good cost amortization over a variety of loads. The constraint index, which acts as a lookup table for attribute names over constraints, is used for the quick detection of attribute names that have no matching constraints. If no attributes match the event can be discarded by the router.

### Implementation

The current Siena implementation is a prototype that consists of Siena servers and client-level interfaces. The C++ version supports the peer-to-peer server and the Java version supports hierarchical servers. Currently, the C++ implementation is not compatible with the Java version. The Siena implementation uses TCP/IP for communication.

### Simulation

The algorithms and topologies used in Siena were examined in a simulated environment. The hierarchical client-server architecture should be used when there are low numbers of parties that subscribe and unsubscribe frequently. The acyclic peer-to-peer model was found to be more applicable to situations where the total cost is dominated by notifications and there are many ignored notifications [CRW99].

### Current and Future Developments

The follow-up project to Siena, Son of Siena, involves the use of XML in representing events and XML routing. In addition Columbia University has developed the XML-based Universal Event Service (XUES) that consists of three main services that support event handling for the Kinesthetics eXtreme (KX) real-time monitoring architecture. The system inputs events using the Event Packager, analyses events using the Event Distiller and dispatches events using the Event Notifier. The system interacts with other event systems using XML, FleXML and Siena.

During the development of Siena-XML interface [Ere00] several problems with translating an XML-based hierarchical namespace to a flat namespace were identified and addressed. In the conversion process the nested structure of XML documents is converted into flat names that preserve the hierarchy by separating the hierarchies with dots. This is a typical way of describing hierarchical content; another would be to use the Windows or Unix file system notation. Now, a problem arises when there are duplicate elements in a hierarchy, which translate to an item with multiple values. Siena does not support this, and the Siena-XML interface currently ignores these duplicate values. One solution would be to include support for wildcards or multiple sets of values, for example simple list objects.

In the future Siena is envisaged to integrate at the network service level, coexisting for example with TCP/IP instead of working above the network level. This would eliminate an extra protocol layer, and provide greater effiency in routing and forwarding. From the Siena viewpoint TCP/IP performs explicit address routing and Siena is based on content-based addressing. The risk in using Siena as a network service is that content-based routing is more computationally expensive than explicit-address or subject-based routing [Ros01b].

There is also work to make Siena support satellite-based wireless communication. Satellite-based communication has desirable properties for transmitting events, because routing is not necessary when the events are broadcasted rather than sent using point-to-point communication lines. Thus it is possible to notify large numbers of interested parties in one hop. However, wireless networking is more unreliable than wireline networking. Moreover, the receiving devices may be different from desktop computers, thus requiring the solution to cope with limited resources.

Siena has also been used as a peer-to-peer network similar to Gnutella. The Java-based Quad uses the Siena prototype and supports query, advertise, and response. One of the differences between Quad and Gnutella is that with Gnutella the messages are propagated to all servers and filtering is performed by the provider at the last step. The main architectural difference between Gnutella and Quad is the separation of clients and servers. Thus the general advantage of peer-to-peer systems in dynamic networking is lost [Hei01].

One of the findings of the Siena project is that expressiveness and scalability are in conflict. Expressiveness is related to flexibility of notification and routing. Scalability, on the other hand, is about vast dimensions, heterogeneity, decentralization, and the use of resources.

### 2.3.9 Elvin

Distributed Systems Technology Centre (DSTC) has developed the Elvin system since 1993 and it has grown from a single person research project to an effort with a team of programmers and researchers. Elvin is a general event notification service, which aims to improve on features identified in a 1995 survey of commercial event filtering software. Elvin started as a publish-subscribe notification service, but currently it is referred to as a content-based routing service. The Elvin team aims to standardize the Elvin protocol through IETF, and the Elvin protocols are written in the style of IETF drafts. DSTC was a contributor to the OMG Notification Service RFP and one of the submitters of the CORBA Notification Service.

Elvin uses a client-server architecture in notification delivery. Clients establish sessions with Elvin servers and subscribe and publish notifications. An Elvin notification is a list of name-value pairs, similar to Siena. Basic primitives are: 32- and 64-bit integer, 64-bit double precision floating point, internationalised string (UTF-8 encoded), or an array of bytes. Subscription expression are defined using logical expressions using a C-like syntax: stock == "abc" && value > 80. The expressions are evaluated with Lukasiewicz's tri-state logic that uses an additional value of indefinite (i.e. true, false, indefinite).

Elvin has language bindings for C/C++, Java, Python, Smalltalk, Emacs Lisp and Tcl. Elvin is content-based, because it allows routing decisions to be made based on the whole message. Elvin features a decoupled security model, in contrast with the traditional point-to-point model, in which communication between publishers and subscribers is authenticated with keys. Producers and consumers can have overlapping key sets. This supports multi-party authorization.

Service discovery is done using a lightweight protocol that is based on multicast. Once a server has been deployed on the network, clients use the protocol to discover the server and dynamically register. Clients also listen to router advertisements, which are also distributed using multicast.

### Clustering

Elvin supports local clustering of servers that improves scalability and distributes the local load. Clustering is used to implement a distributed, but single subscription address space. Routers within a cluster communicate using a reliable multicast protocol over an IP network. An Elvin router may force a client to reconnect to another server in order to reduce load. The Elvin cluster is similar in functionality to a web farm. An Elvin router is a daemon process that runs on a single server and distributes Elvin messages. Each router in an Elvin cluster shares client subscription information with every other node. Not all subscription information is shared; only sufficient information in order for a router to decide if a given notification has any subscribers at any server.

The initial forwarding decision in server-server communication is done based on a list of terms. This hasty approach results in a number of unnecessary notifications at the router level. The Elvin team aims to improve this in the next version of the system.

The Elvin cluster topology consists of a single master router and a number of slave routers. The master router maintains management data. All slave routers listen to management traffic within the cluster and keep information about every node. Routers also keep information about subscription terms of other servers, current states, list of URLs offered by a router for client connection, and current router load and statistics. Master servers listen for join packets and keep track of the cluster as a whole. A new master router is elected using an election protocol if the old one fails.

Messages are first analyzed at a local router and then multicast to the cluster. Each packet must contain the unique router identifier for every node in the cluster, which has a matching term.

Communication between clients and routers employs RPC-style communication with acks and nacks. Delivery has best effort, at-most-once semantics. In the client-server protocol the server may drop notifications, but is eligible to warn the client that it has done so.

#### Federation

There is a different protocol for linking distributed clusters of servers to a federated system. The Elvin federation protocol assumes that the federated topology forms a spanning tree. Moreover, the linking protocol supports the definition of pull filters that constrain the notifications sent to other clusters.

#### Quench

In Elvin terminology Quench means an operation supported by all event producers that gives the producers the possibility to evaluate a subscription expression to cease producing events that are no longer needed and determine which notifications should be produced. In CORBA this would mean that the first event channel refrains from forwarding unnecessary notifications (CORBA does not support client side filtering). The quench is a semantic extension of the subscribe mechanism

In Elvin quench is implemented in the client-server protocol. Any client may request to be notified when the subscription information of the server changes. The client may request information on named attributes in subscriptions. The requested information is sent as an abstract syntax tree. There is support also for an automatic quench, which is implemented in the client library.

#### Mobile Users

Elvin has been extended to support mobile users. One of the requirements was persistence in order to keep undelivered notifications. Elvin is, by design, non-persistent so a prototype proxy was designed to store notifications. The proxy model extends the client-server architecture of Elvin by introducing the proxy as the third component. Proxies act as normal clients to servers, but as a proxy server to clients. In this design, clients connect to these proxies, which mediate the Elvin service [SAS01].

The proxy is able to handle multiple clients with separate sets of subscriptions. Elvin did not support subscription grouping by the client, so support for this was added to the system (the concept of a session). These sessions need not be client specific, but rather they may span multiple clients or applications. This stems from the observation that many people have several devices, but may wish to receive the same set of information regardless of the medium. In order to manage the storage space for undelivered notifications, the proxy supports the definition of a time-to-live (TTL) for each subscription. In addition, clients may specify the maximum number of notifications to keep.

In the current prototype clients explicitly connect to the proxy, and they must connect to the same proxy to retrieve notifications. Proxy discovery and roaming between proxies is not supported. Elvin proxy service is proposed as a solution to proxy roaming, and client migration between networks, however the difficulty lies in that the proxy is stateful entity. Normal Elvin servers are stateless.

#### Non-destructive notification receipt

For users who use many different devices and wish to share notifications, Elvin supports non-destructive notification receipt. This means that the proxy does not destroy a notification upon its successful notification. Elvin ensures that notifications are never delivered to the same client more than once.

Because sessions may contain a number of clients, Elvin supports additional management functionality regarding the set of subscription set by clients. Each client is informed of the current subscription status. There may also be a number of sessions per client, in which case only one notification is sent even if there are multiple matches.

### 2.3.10 Other Event Architectures

#### JEDI

Java Event-based Distributed Infrastructure (JEDI) is a distributed event system developed at Cefriel, Politenico di Milano. In JEDI the distributed architecture consists of a set of dispatching servers (DS) that are connected in a tree structure [CN01]. Each DS is located on a node of the tree and all nodes except the root node are connected to one parent DS. Each node has zero or more descendants. Event subscription and unsubscription requests are propagated by each DS upwards towards the root. Event notifications are processed similarly and forwarded by the local DS to its parent. Upon receiving an event, each DS checks its descendants if they have an interest in the event, and, if required, forwards the event down the tree.

This strategy requires that a given DS knows the event requests of its descendants in order to make the forwarding decision. Moreover, since all requests and notifications are propagated upwards the tree, the communication and processing overhead of the nodes near the root may become a bottleneck. If any of the nodes near the root become disabled, parts of the tree become isolated. In this case the system needs to deal with segmentation and be able to mend the tree or negotiate a new root and a new tree.

A JEDI event is an ordered set of strings, the first string being the name of the event followed by event parameters. An Event Dispatcher can subscribe to a single event or an event pattern. Event patterns are used to filter events, based on parameter matching, for example foo(aa*,bb) matches all foo-named events that have exactly two parameters and the first parameter starts with aa and the second parameter is exactly bb.

JEDI preserves causal ordering of messages, that is, if event e1 caused the firing of event e2, e1 must be delivered first to all interested subscribers. This mechanism allows a pair of components to synchronize through the generation of events [CNF01].

The JEDI architecture is being extended to support mobile clients and ad-hoc configuration [CNP00]. Publish/subscribe middleware is a good candidate for context-aware computing. Asynchronous interest-based communication is a good start for building decoupled and adaptive software components. Compositionality and reconfigurability are being emphasized and JEDI supports mobility with moveOut and moveIn operations. One issue is run-time configuration of the dispatching system, which is also investigated by the JEDI project.

The dispatching servers in the JEDI architecture support mobility by allowing clients to disconnect, move to a new dispatching server and connect while retaining all the notifications. The dispatching servers manage temporary storage for notifications. They also coordinate that no duplicates are received and that the notifications are causally ordered [CN01]. The new dispatching server contacts directly the old one in order to receive the accumulated notifications. The old DS notifies its parent dispatching server to route any further notifications for this client to the new DS.

Notifications are routed in the JEDI dispatching tree from producers to consumers and there is no possibility for adapting the routing strategy to reflect changes in the pattern of communication. The system offers good performance if the tree is organized in a good way that minimizes network traffic. In essence, when clients migrate from one dispatching server to another the load placed by the servers changes. It may be necessary to re-create the dispatching topology to reflect these changes.

JEDI approaches the adaptation of publish/subscribe systems to more dynamic environments by extending the event routing mechanism by adding a new spanning tree routing algorithm. Now, a delegate leader is

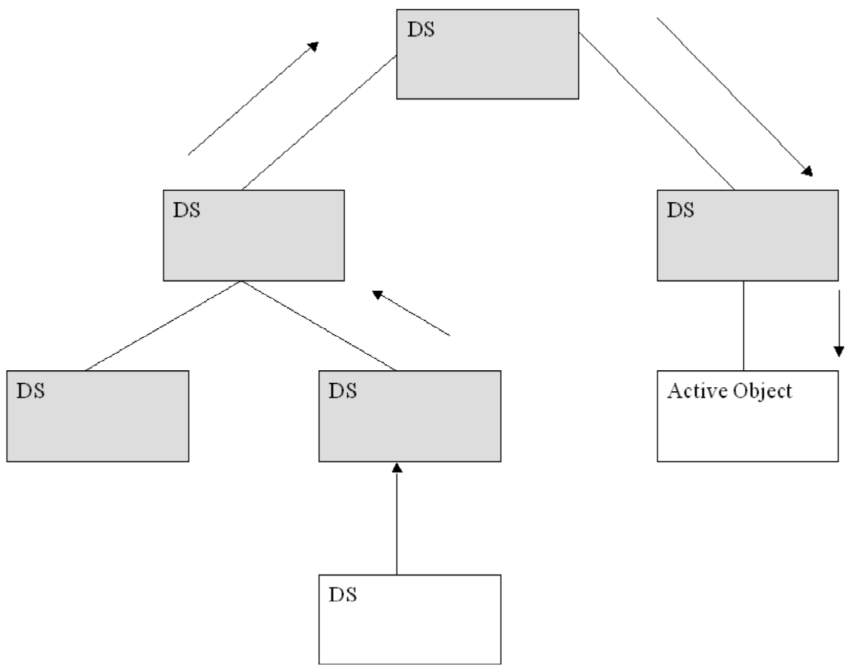


Figure 2.16: Event propagation in JEDI.

responsible for each subscription. The delegate accepts subscriptions of similar type and becomes the leader of the subscribers. It also manages the distribution of the group in the tree. Each dispatcher knows the group leaders for all subscriptions [CNF01].

The JEDI approach is based on dynamically defining the dispatching tree by using approaches similar to multicast routing. The first strategy is to create a minimal spanning tree for each pair of publisher and group of subscribers, but it is considered to be inefficient. The second strategy is to have a single routing tree for each group of subscribers and have different publishers for the same class of events use the same tree.

They use a method called the Core Based Tree Strategy, in which the dispatchers are connected in a possibly cyclic graph and each dispatcher knows its neighbours. Dispatchers broadcast all unique subscriptions to all servers, and all subsequent subscriptions of the same type are sent to the party that sent the original subscription. The original source dispatcher has implicitly become the leader of a group of subscribers and maintains access to that group. Now, the source may balance load by assigning subscriptions to dispatching servers. All dispatchers know all group leaders, and those dispatchers that belong a group know the dispatching tree of that group. When a component unsubscribes, the associated dispatcher either leaves the group, continues to route notifications, or, if it was a leader, the system needs to elect a new leader for that group.

The mobility support in JEDI is still under consideration, for example the latency of updating the dispatching trees when clients are moving very frequently and in the case of abrupt disconnections are still open issues. They are looking at what kind of abstractions we need at lower levels in order to detect disconnections at upper level. The scalability of the JEDI system to Internet-wide use is an open issue. JEDI was used to implement the Orchestra Process Support System (OPSS) workflow management system (WFMS) [CNF01].

The JEDI project ended in 2000 and Cefriel has continued to work on event architectures. They have a project on fault-tolerance and scalability issues in distributed communication based on the publish/subscribe paradigm. They continue to use the JEDI event dispatchers as a reference implementation. The goal of this research is to implement a fault-tolerance JEDI.

Later the JEDI subscription propagation algorithm was improved by introducing advertisements. This new algorithm is similar to the Siena work, and covering relations to optimize routing. The impact of advertisement was evaluated using simulation, and their results show that with advertisement the root node spends much less time processing subscriptions. Their simulation results of 8-85 dispatchers indicate that the processing time of advertisements is quite low (2.65%-2.9%) [BNFT00] [BNT00].

### ECHo

ECHo is a high-performance data transport mechanism that is based on event channels [EBS01]. ECHo uses channel-based subscriptions (similar to the CORBA Event Service). ECHo's derived event channel mechanism implements filtering by adding an application supplied derivation function F to all listeners of a particular event channel and transfering all events that are generated by the sources and passed through the filters to a derived event channel. This scheme resolves issues in the delivery of unwanted events. ECHo is especially optimized for streaming data and data transmission. ECHo has been shown to perform better than Jini (distributed Java events), CORBA Event Channels, and XML-based messaging. ECHo was developed at Georgia Tech and the source is available for academic research purposes.

### Gryphon

The Gryphon system was developed at the Distributed Messaging Systems group at IBM T.J. Watson Research Center. Gryphon is a Java-based publish-subscribe message broker intended to distribute data in real-time over a large public network. Gryphon uses content-based routing algorithms developed at the research center. The clients of Gryphon use an implementation of the JMS API to send and receive messages. The Gryphon project was started in 1997 to develop the next generation web applications and the first deployments were made in 1999.

Gryphon is designed to be scalable, and it was used to deliver information about the Australian Open to 50000 concurrently connected clients. Gryphon has also been deployed over the Internet for other real-time sports score distribution, for example the Tennis US Open, Ryder Cup, and monitoring and statistics reporting at the Sydney Olympics.

Gryphon supports both topic-based and content-based publish-subscribe, relies on adopted standards such as TCP/IP and HTTP, and supports recovery from server failures and security. In Gryphon, the flow of streams of events is described using an information flow graph (IFG), which specifies the selective delivery of events, the transformation of events and the creation of derived events as a function of states computed from event histories.

Information flow graphs contain stateless event transforms that combine events from various sources and stateful event interpretation functions that can be used to derive trends, alarms, and summaries from published events. Each event is a typed tuple. Stateful events depend on the event history. States are used to express the meaning of an event stream and the equivalence of two event streams.

The Gryphon model consists of information spaces, which are either event histories or states. Event histories grow monotonically over time as new events are published. Event sources and sinks are modelled as event histories. States capture certain relevant information about event streams and they are typically not monotonic. Information spaces are defined using information schemas. Dataflows are directed arcs that connect nodes in the graph, which needs to be acyclic [BKS+99].

Gryphon supports four types of dataflows. Select is an arc that connects two event histories with the same schema. Each arc is a predicate on the attributes of the event type in the information space. All events that satisfy the constraint are delivered to the destination information space. The transform arc connects any two event histories which may have different schemas. Each arc has a rule for mapping event types between the two spaces. This rule may include functions that transform particular event attributes. The collapse arc connects an event history to a state using a rule. The rule maps a new event and a current state into a new state. The expand arc is the inverse of collapse, and links a state to an information space. When the state at the source of the arc changes, the destination space is updated in such as way that the sequence of events it contains collapses to the new state. This transformation is non-deterministic.

They have two techniques for the implementation of systems based on

IFGs. First is the flow graph rewriting optimization that allows stateless IFGs to be used with multicast technology. The second is an algorithm for converting a sequence of events to the shortest equivalent sequence of events.

The information flow graph is abstract and separated from the physical topology of the network. The mapping of an IFG to a network of message brokers is nontrivial. Gryphon reduces an arbitrary IFG by rewriting it. All the select operations are moved together and closer to publishers and all the transform operations are also grouped together closer to the subscribers. Transform operations are done at the periphery of the network.

The Gryphon system allows the representation of event histories as states, which is interesting especially for mobile and disconnected users. Wireless users would benefit if a system could inform them with a summary of events that occurred while they were disconnected (the state). The Gryphon system detects failed brokers and reroutes traffic around failed nodes. Moreover, the system incorporates several security mechanisms: access control, and four authentication methods.

Gryphon supports the JMS publish/subscribe API, and supports topic-based subscription. In addition, clients may specify filters using the WHERE clause of SQL92 supported by JMS. Gryphon extends the publish/subscribe one-to-many model with request-reply and solicit-response models. By using unique topics JMS users can use request-reply style messaging. In the solicit-response model a client may make an advertisement to which one or several clients may respond privately.

The basic unit of the Gryphon multibroker configuration is the cell, which is a group of fully connected servers. Cells may be further linked together for geographical scaling through link bundles. Link bundles provide redundant connections between cells, which includes load-balancing and fault-tolerance not provided by gateway-based approaches. The internal protocols and systems ensure that cycles are avoided and messages are routed around failed nodes.

### COM+ and .NET

Standard COM and OLE support asynchronous communication and the passing of events using callbacks, however these approaches have their problems. Standard COM publishers and subscribers are tightly coupled. The subscriber knows the mechanism for connecting to the publisher (interfaces exposed by the container). However, this approach does not work very well beyond a single desktop. Now, the components need to be active at the same time in order to communicate with events. Moreover, the subscriber needs to know the exact mechanism the publisher requires, however this interface may vary from publisher to publisher making this difficult to do dynamically. (ActiveX, COM use the IconnectionPoint mechanism for creating the callback circuit, an OLE server uses the method Advise on the IoleObject-interface). Furthermore, this classic approach does not allow filtering or interception of events [Pla99] [Sri01] [Mic02].

**COM+ Event Service** The COM+ event service [Pla99] [Mic02] is an operating system service that provides the general infrastructure for connecting publishers and subscribers. The service is a Loosely Coupled System (LCS), because it decouples event producers from event subscribers using the event service and a catalog for storing available events and subscription information. In this architecture, an event is a method in a COM+ interface called the event method, and it contains only input parameters.

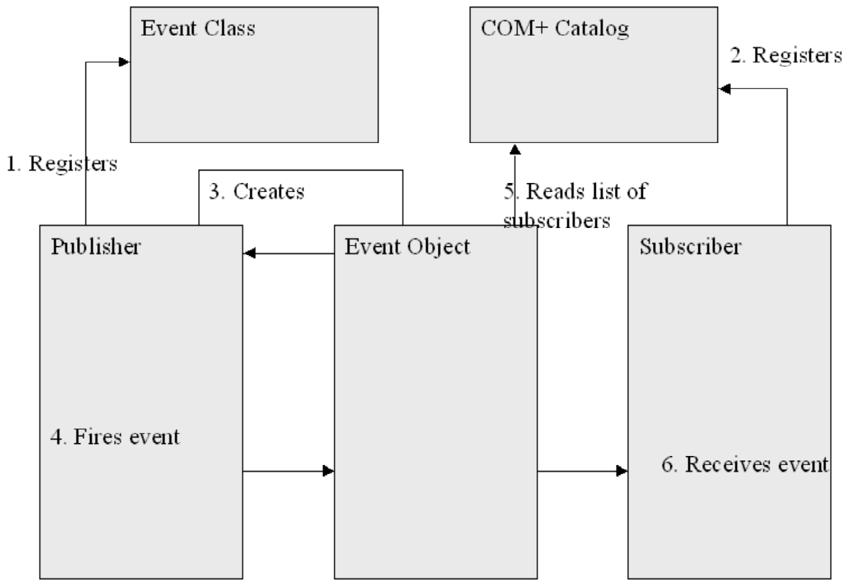


Figure 2.17: The COM+ Event Service.

The following steps are required for producing an event:

1. An event Class is registered.
2. Subscriber registers for an Event.
3. Publisher creates an Event Object at run time.
4. Publisher fires the Event by calling the method in the Event Object.
5. Event Object reads the Subscription List from the Event Store.
6. Delivers the event to the subscriber by calling the appropriate method.

The change in the COM+ Event Service is the addition of the event service in the middle of the communication. The event service keeps track of which subscribers want to receive the calls and mediates the calls. The event class is a COM+ component that contains the interfaces and methods a publisher calls to fire events and a subscriber needs to implement in order to receive the event. Event classes are stored in a COM+ catalog that is updated either by the publishers or by administration.

Subscribers register their wish to receive events by registering a subscription with the COM+ event service. A subscription is a data structure that contains the recipient, event class and which interface or method within that event class the subscriber wants to receive calls from. Subscriptions are also stored in the COM+ catalog either by the subscribers or by administration. Persistent subscriptions survive restarting the operating system, transient subscriptions will be lost on restart or reset.

The publishers use the standard object creation functions to create an object of the desired event class. This event object contains the event system's implementation of the requested interface. The publisher then calls the event method that it wants to fire. The event system implementation of that interface looks in the COM+ catalog and finds all the subscribers who have expressed interests in that event class and method. The event system then connects to each subscriber, using direct creation, monikers, or queued components, and calls the specified method. Event methods return only success or failure. Any COM client can become a publisher and any COM+ component can become a subscriber.

The current event system has several limitations. The subscription mechanism is not itself distributed and there is no support for enterprise-wide repository. Secondly, event communication in the system is done either by DCOM or Queued Components, which are both one-to-one communication mediums. The delivery time and effort increases linearly with the number of subscribers, which means that the system is not scalable to firing events to many subscribers.

However, client-side disconnection is supported with queued components. COM+ supports components that record a series of method invocations (event occurrences) and are able to play them back in the recorded order. These components can be distributed using messages. Since the event object may be defined as queuable, a disconnected client may playback the desired event object upon reconnection.

COM+ Events can be extended to support filtering, which needs to be implemented either on the publisher side or on the subscriber side. If an event is filtered by a component on the publisher side, it is never delivered to the event service. If an event is filtered on the subscriber side the event service will make the decision of whether to deliver the event to a particular subscriber [Mic02].

Filtering on the publisher side is done by attaching a filter object on the event object interfaces (which correspond to events). The filter may query the subscription information and, for example, change the firing order for a set of subscribers. The subscriber-side filtering is done using parameter filtering for each subscription and method invocation. Parameter filtering evaluates the subscription FilterCriteria property against the parameters of the event method. The filter criteria string recognizes relational operators, nested parenthesis, and the logical keywords AND, OR, and NOT.

**Interoperability with .NET** The COM+ Event System needs to generate some metadata in order to interoperate with the .NET world. However, an abstract definition of the Event Interface, Event Classes, and their attributes is needed [Kis01].

**.NET** The .NET framework supports events at many levels. There is support for programming language level events, interoperability with COM events. The interoperation of Visual Basic .NET code and legacy COM component events is done using a runtime callable wrapper (RCW). In VBN listeners create event handlers, which are added to sources. The connection between events and event handlers is implemented by special objects called delegates. The benefit of the .NET runtime is that the events from components written in different languages, say C# and VB, are interoperable.

Microsoft's messaging infrastructure is called Microsoft Message Queuing (MSMQ) [Mic99]. In this kind of architecture, applications receive and send messages using queues. MSMQ supports disconnected operation and is especially useful in intermittently connected Windows CE/PocketPC devices. MSMQ allows application writers to asynchronously send messages. MSMQ CE version can, for example, be used for

- Messages transferred when in range (delivery tracking, quality control)
- Messages transferred once in a while (intelligent set-top boxes, inventory control, . . . )
- Producer and Consumer are not active at the same time

### MSMQ Product Architecture

MSMQ queues are either private or public. Public queues are stored in a directory service called Message Queue Information Store. Public queues are more expensive to use, because directory access is not free. Moreover, Windows CE clients cannot host public queues. The CE MSMQ independent client can operate independently if the server is unavailable and store messages locally. The servers route and store messages and support clients in the form of client proxy server and queue manager. On the other hand,

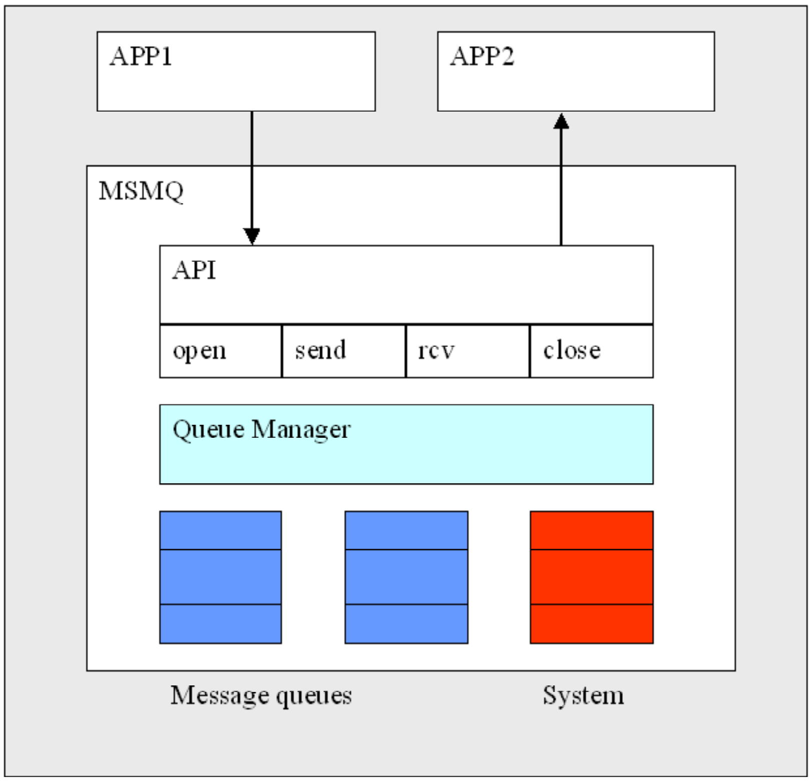


Figure 2.18: MSMQ Product Architecture. The Queue Manager connects to other Queue Managers in order to communicate between different hosts.

MSMQ supports also dependent clients that cannot store local messages and need the server. The architecture supports three delivery options. Fast memory-based reliable store and forward supports network loss, but not reboot and cannot guarantee exactly-once semantics. Persistent guaranteed store and forward supports reboot, and persistent transactional message queuing guarantees exactly-once in-order delivery. Transactional guarantee at commit time is about delivery to the local queue. In essence, the system supports local all-or-nothing guarantee [Mic99].

The MSMQ Windows CE-version (2.12+) supports roaming and dynamic adapter switching. It tracks Network Interface Cards (NIC) and restarts immediately after reconnection. The transparent storage is based on one queue per file. The footprint of the system is around 100-150K. The CE implementation has several limitations: clients must use direct names, supports only private queues, the routing is limited, transactions are not supported (once and in-order are supported), no system support for encryption or ACL and no remote queue access. The system can be deployed in a client-server or client-client environment and also for message-based IPC within a device.

The next version of MSMQ, Message Queuing 3.0, is available in Windows XP and supports messaging over the Internet, one-to-many messaging model and message queuing triggers [Mic02]. HTTP is supported as an optional transport protocol and an XML-based SOAP extension is introduced that defines a reliable end-to-end messaging protocol. MSMQ is by default based on a proprietary TCP-based protocol. The system also supports real-time messaging multicast using the Pragmatic General Multicast (PGM) protocol [IET02]. This protocol supports only an at-most-once quality of service and does not support transactional sending. The MSMQ 3.0 programming model is extended to allow an application to send a single message to a list of destination queues.

Message Queuing Trigger is a service that allows an application to assign functionality in a COM object to be triggered when a message arrives in a particular queue. Each trigger is associated with a queue and applies a set of rules for every message arriving in that queue. An action is executed when all conditions in a trigger hold [Mic02].

Message routing is done using the lowest-cost route that is available. If a network fails, the next lowest-cost route is used to deliver the message. Administrators define costs for each network with the management software (MSMQ explorer).

#### Websphere MQ

IBM's MQSeries, currently known as Websphere MQ, is one of the most popular MOM products for electronic business. The product supports heterogeneous any-to-any communication between 35 different platforms. MQ is compatible with JMS and integrates with Java Beans 2.0 (EJB), XML, and JSP framework and servlets. MQ also supports SOAP for web service creation. JMS 1.0.2 compliant embedded JMS provider supports point-to-point and publish-subscribe messaging [IBM02b].

MQSeries Everyplace enables access to enterprise data and supports mobile workers. Everyplace is available for a number of platforms, for instance Linux, WinCE, EPOC, and PalmOS. The PDA type messaging is similar to messaging for other platforms with queue managers. A queue manager manages queues that store messages, and applications communicate with their local queue manager. Remote queues are owned by remote queue managers, and each message that is inserted into a remote queue gets transmitted over the network. The queue manager may support a local queue, in which case the client is capable of supporting asynchronous communication. If no local queue is present, the client is bound to synchronous communication. Another configuration option is whether the client supports bridges, and is capable of exchanging messages, with other MQSeries queue managers.

A typical client-server configuration is a scenario where a server hosts the queue manager and clients connect to it with a bi-directional communications link (with a proprietary MQSeries protocol). The client infrastructure is quite lightweight, because it is dependent on the server queue manager. In a multi-server scenario, clients employ message channels, which support unidirectional, safe, and asynchronous message exchange. Channels are a form of end-to-end service provision and consist of the source queue manager, a number of intermediate managers, and the destination queue manager. The footprint of the system is 64K for Palm and 100 K for a class file with Java devices [IBM02b].

### 2.3.11 Discussion

The following list gives an overview of the discovery and event delivery mechanisms used in the event and message architectures presented in this paper.

**CORBA Event Service** Discovery method is unspecified. Centralized component for event propagation (Event Channel) that provides anonymity.

**CORBA Notification Service** Discovery method is unspecified. Centralized component for event propagation (Event Channel) that provides anonymity. Components may discover the event types and subscribers in an event channel.

**Siena** Access nodes advertise services (advertised/unadvertised). Four different server topology configurations: client-server, hierarchical, acyclic peer-to-peer, and generic peer-to-peer. Events are routed.

**JMS** Discovery using JNDI. Messages are published to queues or topics. Not a full event service.

**Cambridge Event Architecture** Discovery method is unspecified. Direct notification, filtering may be also implemented by a mediator. Also support for CORBA Event Service.

**Java Distributed Event Architecture** Does not specify an interest registration service. Direct notification, the generator knows the listeners.

**JEDI (Java Event-based Distributed Infrastructure)** Subscriptions processed by the infrastructure. Event Dispatcher is a centralized component that supports event subscription and unsubscription. The distributed implementation of system consists of a number of EDs connected into a tree.

**Gryphon** Subscriptions processed by the infrastructure. Content-based routing through large-scale public networks. Uses JMS API for clients.

**ELVIN** Discovery of servers using multicast/unicast within a cluster. Infrastructure manages subscriptions. Message routing, client-server and server-server protocols. Different protocol for inside clusters and between clusters.

The following listing presents the support for wireless and mobile clients in the architectures:

**CORBA Event Service** Partial support for disconnected operation. Events are stored at Event Channels.

**CORBA Notification Service** Partial support. Events are stored at Event Channels, QoS, possibility for querying status.

**Siena** No mobility (mobility has been discussed; an extra layer). Wireless (satellite) support under work.

**JMS** Durable subscriptions support intermittent clients.

**Cambridge Event Architecture** Limited. Disconnected operation possible using a proxy.

**Java Distributed Event Architecture** No, supports mediators (proxies).

**JEDI (Java Event-based Distributed Infrastructure)** JEDI supports message buffering and mobility with two commands: move_out and move_in. Further support is under work. Load balancing through subscription groups. In the future, ad hoc networking.

**Gryphon** Support for disconnected users (buffering). Support for event history collapsing (summarization).

**ELVIN** ELVIN has been extended to support disconnected clients using a proxy. Clients connect to the proxy. No mobility between proxies.

**MQSeries, Websphere MQ** Compatible with JMS. Supports disconnected operation (buffering).

**MSMQ** Buffering. Supports the changing of the network card etc.

## 2.4 Conclusions

Message oriented middleware and event notification are becoming more popular in the industry with the advent of the CORBA Notification Service, the Java Messaging Service, and other related specifications and products from many vendors. Many research projects have addressed and are addressing issues of scalability, compound event detection, mobility, and fault tolerance, to name a few topics. There are many ways to classify event systems, and many possibilities for their use depending on the requirements.

Traditional MOM systems are getting influences from event-based systems. For instance, JMS supports both queues and publish-subscribe style communication with filtering. However, these systems usually lack support for distributed coordination in notification delivery and employ topic-based routing. Current event systems are evolving towards content-based routing, which uses the whole notification as an address. In content-based systems clients can change their interests without changing the addressing scheme (adding a new topic).

Scalability has been emphasized in Siena, and it has been designed for Internet-wide scalability and tested in a simulation environment with various network topologies. However, scalability introduces latency, which creates problems for notification semantics and mobility. Other systems tackle scalability and fault tolerance by creating clusters (Elvin) or cells (Gryphon) that contain connected servers. These clusters are connected using point-to-point links and possibly different protocols. Multicast and fault tolerance can be provided within the clusters. Event systems are logically centralized, however the CORBA Event Channel is also physically centralized, creating a possible bottleneck.

Ad hoc networks are emerging with the introduction of short-range radio communications. Ad hoc event systems support the dynamic addition and removal of event servers (or event dispatchers). However, ad hoc event topologies are currently an emerging research topic and some research issues have been raised in JEDI.

From the mobile Internet and ubiquitous computing view point JEDI and Elvin have examined a more thorough support for disconnected operation. JEDI supports both mobility and disconnected operation as a service, and Elvin only disconnected operation (with a few additional features) as an extension to the original architecture. Almost all message queue-products support disconnection with various semantics. One important decision is whether to include this support as an extension or as an integral part of the event service. If fault tolerance and at-most-once semantics (or roaming) are to be supported, it may be necessary to integrate this functionality at the service level. Another open issue is whether event service should reside at the network level or at the application level (OSI stack). For Internet scale routing, as proposed in Siena, it might be beneficial to have some support at the network level.

Only a few architectures support complex compound event filtering. Usually event filtering is done using simple parameter wildcard matching (JEDI), simple clauses (COM+, Elvin) or SQL (JMS, Gryphon), and Extended TCL (CORBA). Compound event detection is supported in CEA with event templates and in Siena by detecting a sequence of simple filters. Compound event detection is also a feature that may be integrated as an external component or within the infrastructure.

Many systems do not consider the process of locating and connecting producers, or locating event channels (Notification Service). Some architectures, such as Siena, JEDI, and Elvin support this within the infrastructure. There are two completely different problems: one is locating an access point using multicast or unicast to a known address, and another is to use either the infrastructure or some other service to locate channels or subscribe. In systems such as Siena and JEDI, subscribing is accomplished by using simple string-format requests. With CORBA it is necessary to obtain an event channel and go through a more complicated procedure in order to obtain references to proxy objects. This process of obtaining the event channel reference according to interests is not specified.

Many message queue products are supporting XML-based solutions, such as SOAP, as one of the transport options. MQSeries, MSMQ, and .NET support SOAP and Siena has XML bindings as well. XML has many applications in messaging and event-based communication. XML can be used to define the content of messages. For example, JMS facilitates XML-based messages and the routing of XML.

However, the building blocks of the semantic web, such as ontologies, are not yet supported. Ontologies and XML-derived languages could well be used to define events and event systems, and improve interoperability. XML and a suitable ontology would enable the specification of complex event monitoring tasks that are uploaded to routers or, for example, web services. In the future, it is envisaged that event applications have policies (specified in XML, for example) for event semantics, buffering and other information that affects the delivery of notifications.

# Chapter 3

# XML Protocols

Services provided over the Internet are becoming increasingly a major part of the current world. Currently the most often used system for implementing these services is a Web browser sending its service requests to Web servers, which then generate the responses dynamically. This system leaves much to be desired as anything more complex has to be done with add-ons such as cookies. Therefore the need for a more flexible and powerful system is obvious.

The client-server, Request-Response paradigm described above contributes much to the inflexibility of current service models. A new general architecture and a protocol to go with it are needed, if new service models are required. This system should also be simple to implement and provide enough flexibility to allow rapid development and deployment of various services.

We will review the concepts of XML and Web Services and then concentrate on the protocol seen as the basis of these services. We will also discuss issues related to these new services in wireless environments and go through some proposed solutions.

## 3.1 XML

Currently the standard way of marking up document structure on the World Wide Web (WWW) is Hypertext Markup Language (HTML), which is based on Standard Generalized Markup Language (SGML) (the technical term for HTML is an SGML application). SGML is a framework for creating markup languages. This is done by writing a Document Type Definition (DTD), which describes the allowed markup tags and their syntax. SGML documents are typically hierarchical, i.e. elements (content between a start tag and its corresponding end tag) contain other elements in addition to text.

The intended way for an application to parse an HTML document is to implement a full-blown SGML parser, which uses the HTML DTD to parse each element according to its defined syntax. However, SGML allows DTD writers to leave certain parts optional, which is useful for both cutting the size of documents (e.g. by omitting end tags for specified elements) and for decreasing the ratio between markup and actual content. This makes writing an SGML parser difficult, and so WWW browsers typically implement only an HTML parser.

The World Wide Web Consortium (W3C), aware of the problems with HTML being an SGML application, set out to simplify SGML. The result of this simplification is now known as Extensible Markup Language (XML) [W3C00a]. XML does not allow implicit content: all start tags must have a corresponding end tag. In addition, XML does not contain some rarely-used features of SGML.

```
<?xml version="1.0"?>
<message status="urgent">
  <from>Boss</from>
  <subject>Reports</subject>
  <text>
    I haven't yet received those reports you promised
    to deliver yesterday.  I need them ASAP.
  </text>
</message>
```

Figure 3.1: An Example XML Document

An example XML document is shown in Figure 3.1; tags are limited by <>, end tags begin with /. The part between a start tag and its corresponding end tag is called an element and the material strictly between them is called the element's content. Each element except the root (`message` in the example) is contained in another element, called its parent element. This contained element is naturally called a child element of its parent. An element may contain attributes inside its start tag (`status="urgent"` in our example). These attributes typically affect the processing of the element in some application-defined way.

As with SGML, specific markup languages can be created with XML using a DTD. However, DTDs are not seen as a good fit for XML so there have emerged a few alternatives. The most visible of these is XML Schema ([W3C01b] and [W3C01c]), a W3C Recommendation. XML Schema syntax is XML instead of the DTD language, so it seems to be a better fit for XML data. Also, XML Schema allows a more fine-grained and flexible approach to restricting element content. DTDs are still expected to last for a while, though, mostly due to their already-familiar syntax and established use base.

## 3.2 Web Services

The term Web Services has in recent times risen to prominence. The point of Web Services is to unite a large variety of different platforms into large distributed systems using simple, standardized protocols and interfaces. XML is an important component of Web Services as they are realized today.

The definition of a Web Service is not very clear-cut. However, there are certain overall characteristics that fit into all used definitions. The central one of these is the use of XML practically everywhere. XML is used to describe the service interfaces, to locate services and even to encode the actual messages. XML is beneficial since it is flexible, standardized and popular.

The W3C has also embraced the Web Services area with its Web Services Activity, which was started in January 2002 as an extension to the XML Protocol Activity. This activity consists of the Architecture, Description and XML Protocol Working Groups. The other Working Groups besides the XML Protocol one are still in their infancy, but the Description one has a basis in the Web Services Description Language (WSDL) specification [W3C01g]. WSDL does not define a new syntax but rather reuses XML Schema syntax, since that provides a good fit for also describing interfaces.

It is expected that in the near future the number of mobile devices having a continuous wireless connection to a network will increase rapidly. Because of this, it is important to evaluate the various Web Service technologies in light of the unique challenges offered by mobility. Of the three parts of Web Services, protocol, description, and discovery, the most obvious part to concentrate on is the protocol since that one most clearly is affected by the move to wireless networking. In addition, if server components can be on a mobile platform, discovery will be more complicated than with fixed services, though this will probably be handled with special addressing schemes suitable for mobility rather than by changing the actual discovery process.

## 3.3 Protocols

The W3C maintains a table of various XML protocols and their feature lists. This is a pretty diverse collection; the only unifying aspect of these protocols is that they use XML as their messaging format. Of these protocols, the most interesting for different messaging purposes is W3C's SOAP ([W3C01e] and [W3C01f]), which is also used as a basis for some other mentioned protocols.

Below we will be concentrating on SOAP. However, the emerging issues are mostly related to XML use as a message format and our considerations should be applicable for other XML-based messaging systems.

### 3.3.1 History

The origins of SOAP lie with UserLand Software. Their Frontier product is a content management system for the WWW. It also includes a Remote Procedure Call (RPC) interface to more easily provide interconnections between various services on the WWW. For this RPC system, they designed a new protocol, XML-RPC [Win99].

XML-RPC is a minimal protocol, intended to be used only for simple RPC needs. There are provisions only for a single Request-Response round trip messaging, there are only a few basic datatypes and the only transfer protocol is Hypertext Transfer Protocol (HTTP). These can also be seen as advantages of XML-RPC. Due to its simplicity, it is easy to implement and there are in fact dozens of implementations in several different languages. There are also no provisions for extensions so the specification has remained stable for several years.

Even though other communication patterns can be built on top of a RPC framework, this would require careful specification of interfaces and messaging semantics. This led to the need for an extensible messaging system. Microsoft had expressed interest in utilizing XML-RPC in their future products, so they teamed with UserLand to produce such a system, which was named SOAP, for Simple Object Access Protocol.

SOAP gained popularity quite fast after its initial launch and other companies joined Microsoft and UserLand in developing SOAP further. Version 1.1 was published as a W3C Note [W3C00b] in May 2000 by Microsoft, IBM, Lotus, DevelopMentor, and UserLand. After this, SOAP was adopted by W3C's newly formed XML Protocol Activity for standardization. This activity has published Working Drafts of version 1.2 of SOAP during 2001, the latest one being from December. Recently this activity was retitled the XML Protocol Working Group inside the newly formed Web Services Activity.

### 3.3.2 Features

We will be concentrating on the features of SOAP version 1.2. There may be a few points where we might mention that a feature first appeared in this version.

The SOAP message structure is shown in Figure 3.2. A SOAP message is an XML document with the root element being Envelope. This element contains one or two elements, the first of these being an optional Header and the second one being a mandatory Body. The Header may contain any number of child elements, called Header Blocks. The SOAP specification

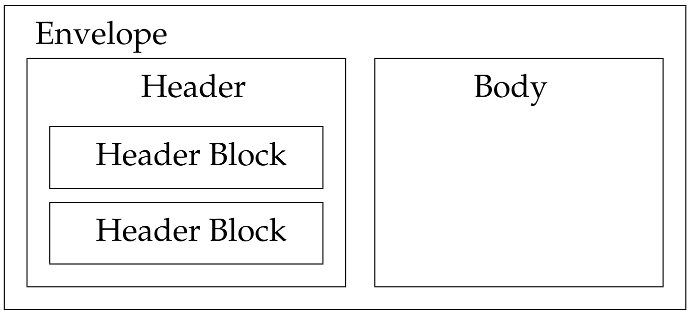


Figure 3.2: The Structure of a SOAP Message

does not address the content of the Body element in any way, other than requiring it to be well-formed XML and specifying its structure in the case of errors, so-called SOAP Faults. This leaves it to each application to define how the content of the Body element is to be interpreted.

The extensibility of SOAP stems from the fact that the content of the Header element is very loosely specified. There are practically no requirements on the type or content of individual header blocks. The only semantics that are defined for header blocks are a few optional attributes describing the encoding of the block, the intended recipient of the information in the block, and an indication of whether the block's semantics must be understood by the recipient. This loose specification allows application developers to freely define their own header blocks with appropriate semantics.

SOAP's extensibility allows it to be used in a wide variety of situations. There is in fact no necessity for SOAP messages to be responded to: the specification assumes only a one-way message transfer from the sender to a receiver. However, as the W3C's XML Protocol Usage Scenarios document [W3C01a] describes, by defining a few header blocks, SOAP can be used to support such communication patterns as Request-Response, RPC, Event Notification and Conversation. In addition, there are provisions for independent intermediaries to be placed on the path between the initial sender and the ultimate receiver.

SOAP version 1.1 was still practically tied to HTTP as a transfer protocol. There was no other specified protocol mapping nor was there a suitable protocol framework to assist in using other protocols. This has changed in version 1.2, where the HTTP binding was removed to the Adjuncts section of the specification and replaced in the main part by a generic protocol binding framework. This framework should make it easier to use SOAP over non-HTTP transfer protocols, and in fact there already are implementations of SOAP that support other protocols. In particular, the XML Protocol Working Group is attempting to specify a SOAP binding for email to illustrate the usefulness of the binding framework.

There has been only very little SOAP-specific work done on security aspects of Web Services. Currently the only way of having some security with SOAP is to use Secure Sockets Layer (SSL), as is done with HTTP. However, the Web Services world is much more complex than the plain Web world and SSL does not address all relevant issues such as authentication in connection with intermediaries and third parties, or security after a message has reached its destination.

The security situation is changing now that people realize SSL is not sufficient for the needs of SOAP. The W3C is working on encryption, signatures, and key management in the context of XML and the results of these efforts should be easily applicable to SOAP also. There already is an application of XML signatures to SOAP published by Microsoft and IBM as a W3C Note [W3C01d].

### 3.3.3 Current State

As mentioned above, SOAP is now officially being developed by the W3C. It is still at the Working Draft stage, which means that the Working Group is still making modifications. Version 1.2 seems to be quite stable, as many of the still-open issues only require clarification and in most cases discussion has converged to an acceptable solution.

The XML Protocol Working Group was originally chartered to be disbanded in April 2002. However, this timetable would have required the Working Group to publish a Candidate Recommendation in April 2001 and a Recommendation in September 2001. Since SOAP is still at the Working Draft stage, it seems that the timetable will slip by at least a year, with a Recommendation expected to be published in late 2002.

SOAP's roots in XML-RPC are also somewhat of a hindrance to full utilization of SOAP's features. XML-RPC is a RPC protocol over HTTP, as was SOAP in the beginning. However, the current version of SOAP is neither RPC- nor HTTP-specific. Even so, people often associate SOAP with these two concepts and this misconception also causes misunderstandings of the SOAP specification.

### 3.3.4 Implementations

SOAP has indeed become popular in recent times. There are already dozens of implementations conforming to the SOAP 1.1 specification, available in all popular, and also some less popular, programming languages for all common platforms. Apache, the leading Web server, has a full implementation and newer Web browsers also contain client-side functionality.

There are also several SOAP toolkits available for various languages. The most popular in the Open Source world is undoubtedly the `SOAP::Lite` module for Perl. Microsoft also includes its own SOAP toolkit in their .NET development framework. There are also SOAP bridges so that CORBA or COM objects can be exposed as Web Services. UserLand Software maintains a comprehensive list of SOAP 1.1 implementations.

Interoperability testing of the various implementations is hampered by the fact that there is no good test suite. Therefore interoperability can only be expected in common cases and more obscure parts of SOAP probably do not get much interoperability testing. The XML Protocol Working Group will change this by designing comprehensive conformance requirements and a test suite to go with these requirements. However, this work is still in an early stage.

Current SOAP implementations only conform to SOAP version 1.1, and, as mentioned above, the lack of a conformance suite precludes exact determination of how conformant various implementations are. It is understandable that implementations of version 1.2 have not appeared, since it is still at Working Draft stage and may change before becoming a Recommendation. The XML Protocol Working Group keeps a partial list of 1.2-conformant implementations and their statuses. It is expected that most vendors will release 1.2-conformant implementation after the Recommendation has been issued.

## 3.4 XML over Wireless

If XML Protocols are intended for use in the services of the future, the needs of mobile users must be taken into account. Mobile users are typically behind low-bandwidth wireless links and the protocols and data formats originally designed for wired networks may be too heavy for wireless connections.

### 3.4.1 Problem Areas

There are several problems in trying to use SOAP over a wireless connection. The most obvious of these is that XML documents tend to be quite large, since the tag names are usually quite descriptive and XML does not allow certain redundant information to be left out. Also, the typical underlying transfer protocol in SOAP implementations is HTTP, which might not be suitable for typical wireless environments.

### 3.4.2 Different XML Protocols

One solution to problems with SOAP would of course be to abandon SOAP completely in favor of a simpler protocol. A common alternative would be XML-RPC, whose messages are quite a bit smaller than the typical SOAP message, since XML-RPC does not concern itself with extensibility like SOAP does.

Switching to XML-RPC would naturally incur some losses also. XML-RPC is tied to HTTP, which, as mentioned above, might not be suitable. The only communication pattern available with XML-RPC is RPC with a very simple type system. While careful specification of interfaces would permit implementation of other communication patterns, SOAP has the advantage of having these patterns built in.

Another problem with using a non-SOAP protocol would be that the majority of Web Services in the future is expected to understand only SOAP, since multi-protocol Web Service implementations are not very common. This would necessitate the creation of XML-RPC-to-SOAP bridges and probably also specifying the internal workings of these bridges.

### 3.4.3 Transfer Protocols

The only transfer protocol currently specified for SOAP is HTTP, which is practically always used over TCP. However, TCP is not very well suitable for wireless links and HTTP itself is somewhat heavy. Of the implementations, `SOAP::Lite` for Perl supports several protocols other than HTTP including FTP, raw TCP and Jabber, an instant messaging protocol. Microsoft's .NET also supports an instant messaging protocol.

A new protocol framework, Blocks Extensible Exchange Protocol (BEEP), was published by the Internet Engineering Task Force (IETF) in March 2001 as a Request for Comments (RFC) [Ros01a]. BEEP is a peer-to-peer protocol that supports connection sharing between logically separate sessions. There already exist implementations of BEEP for Java, C and some other languages. In some circles, a suitable BEEP-based protocol is seen as a possible replacement for HTTP.

The Internet Engineering Steering Group (IESG) has recently approved a proposed SOAP mapping on top of BEEP [OR02] to be published as an RFC as soon as coordination with the Internet Assigned Numbers Authority (IANA) results in assigning standard numbers for profiles and such. There is also work underway in mapping BEEP on top of Stream Control Transmission Protocol (SCTP), which is expected to be a popular transport-layer protocol in wireless environments.

### 3.4.4 Compression

Compression of XML documents sent over the network would seem to be the method that gives the largest payoffs. There are three different ways to approach compression: non-XML-specific methods, methods taking advantage of XML's inherent structuring and binary XML, which preserves the document structure even in compressed form.

### Generic Compression

Generic compression algorithms can naturally be used also for XML documents. They are to be expected to perform well due to XML documents being text and the element parts being highly repetitive.

Generic compression is already publically available in SOAP implementations. For example, the popular `SOAP::Lite` module for Perl implements transparent deflate compression using the zlib compression library. In addition, the popular Apache web server has an extension module (not SOAP-specific) for zlib compression of served documents.

Typical compression algorithms achieve good compression ratios by exploiting redundancy in the data. From this it follows that they perform better on larger documents. While XML in general might be used for even very large data collections, the individual SOAP messages are quite small, typically well under 1KB. Therefore the compressor may perform badly, possibly leaving the compressed size to over 50% of the original size.

### XML Compression Methods

Since XML is as popular as it is, it is to be expected that XML-specific efforts are also made, in compression as in other fields. XML-specific compressors typically exploit the additional structure present in the data. Usually these compressors can also exploit DTDs and XML Schema definitions of the document structure to achieve even better compression.

Two well-known XML-specific compressors are XMill and XMLZip. Of these, XMLZip seems to be defunct as its creator, XML Solutions was acquired by Vitria, who have not included XMLZip in their product line. There are also other compressors, such as ICT's XML-Xpress compressor, but these compressors are typically proprietary and may depend on patented algorithms.

Let us now take a closer look at the XMill compressor as its working principles are publically available. The XMill compression is based on splitting the XML document into a structure stream, which contains the tags and at least one, possibly more, content streams. Each of these streams is separately compressed with deflate compression.

Now, XMill achieves a better compression ratio than generic compression, since the structure stream containing all the tags has more redundancy on its own than as a part of the whole XML document. In addition, by grouping element contents that are expected to be similar (such as, in the case of RPC, all integer parameters) into their own content streams, the content streams also become highly redundant and susceptible to efficient compression. In addition, XMill allows users to specify the structure of the content inside elements to achieve even better compression (such as compressing generic dotted-decimal IP addresses to the theoretical limit of four bytes).

XMill also does not require DTDs to be available for the data like some other XML compressors. This is a benefit since SOAP prohibits the use of DTDs for any part of the data. However, this benefit should disappear as the XML community moves away from DTDs and toward XML Schema definitions, which are also the basis for WSDL, the interface description language for Web Services. A drawback of XMill, like of other XML compressors, is that the methods require quite a bit of data, and can therefore be even worse than generic compressors on small message sizes, such as we have with SOAP.

**Binary XML**

A specific type of XML-specific compression is binary XML, which is different enough to merit separate treatment. In binary XML tags are replaced by binary tokens, reducing each tag to one or two bytes. Standard attributes of elements can also be tokenized in this way. The benefits of using a binary encoding also manifest themselves even on shorter messages, like those used in SOAP. In addition, it is also possible to compress the content of elements independently of the tag compression.

Another benefit of this tokenization is that the document can be parsed directly from the compressed form without having to uncompress first, since the original structure remains intact; only the tags are changed. Also on the sending end the sender could generate binary XML directly. These could be beneficial since handling strings, which regular XML requires, is more time-consuming than handling pure binary data.

One well-known format of binary XML is WAP Binary XML (WBXML). This was published by the WAP Forum as a W3C Note [W3C99] for the needs of Wireless Application Protocol (WAP), which needs to be suitable for small devices with wireless connections. In WBXML the tokens are divided into code spaces and during encoding/decoding there is a default code space at each point in processing.

There is already further work done based on WBXML. The best-known is Millau [GS00], which extends WBXML to more efficiently encode certain common data types such as integers. Since Millau is also an encoding system, it also assigns the binary tokens into code pages in a way that should give a good compression ratio. In addition, Millau compresses the string data (both the string table and the element content). In experiments, Millau reduces document sizes to 20% of the original. This ratio stays constant, unlike with gzip, which improves as the document size increases. Because of this, Millau appears to perform worse than gzip for document sizes over 5KB.

The Millau system also includes APIs for processing the encoded format directly without needing to convert it to XML first. This provides a clear speedup even in the cases where the parser is call-compatible with traditional XML parsers, i.e. the parse events or tree it generates have the actual tag names instead of tokens. By having the parser use only tokens further speedup is achieved.

## 3.5 Summary

Judging from industry support, the deployment of Web Services is about to increase quickly in the near future. With increasingly more sophisticated and powerful mobile devices coming to market, the number of users in wireless environments wanting to use these services seems likely to also grow rapidly. Reconciling the protocol overhead of Web Services with the still quite limited data transfer capabilities of wireless devices is therefore very important.

One option would of course be to do nothing and accept the overhead of Web Services as a natural part of communication, even in the wireless world. This attitude reputedly works for some people. The obvious benefit would be that there would be no need to implement a separate solution for wireless devices. Instead, any Web Service implementation would suffice for both the wireless and the wired worlds. This option should be kept in mind as a baseline due to its simplicity, but tests should be done comparing it to various proposed improvements.

Switching from SOAP to another Web Service protocol would not be very beneficial, as this would require customized bridging solutions with message translation. But switching to another transfer and lower layer protocols could be used with ordinary SOAP intermediaries that understand the transfer protocols. There would in this case be no need to touch the messages except insofar as SOAP intermediaries normally do while processing a message. The wisest course here would be to identify solutions that are expected to become popular, so that there would be fewer problems with support. Currently SOAP over BEEP appears to have the most momentum, with BEEP running on top of either TCP or SCTP.

Generic XML compression schemes should also be ignored as they typically perform worse than more generic compression on SOAP messages. Here the deflate compression method, which is expected to be supported quite widely, should be kept as a baseline, and proposed other schemes compared with it. Some form of binary XML would seem like a better alternative and should be investigated. Millau could be a suitable base for further work; especially SOAP-specific token codes could be assigned. In addition, extending the format with a caching system would remove the need of sending a message-specific full string table with each message, since it is quite possible that several messages use the same element types.

# Chapter 4

# Synchronization

## 4.1 Introduction

As stated in the introduction of this document, one of the long term goals of the project is a mobile distributed information base (MDIB) especially suited for XML storage, with characteristics such as high availability, consistency and support for weakly connected and disconnected operation. In this chapter we will review research relevant to this topic, with focus on *data synchronization.*

Data synchronization (see e.g. [Syn00]) is traditionally understood to be the process of making two sets of data look identical. An exact definition of the term seems to be lacking (the term is not included common computing dictionaries), and it is used in slightly different meanings depending on the context. Here, the term will have the following meaning:

**Data synchronization** Assume two sets of data that have some parts in common. Data synchronization between the sets is the process of making the common parts identical, after changes have occurred in either or both sets. The synchronization process should not ignore changes made to the common part in either set.

As an example of data synchronization, as defined above, consider two data sets $S_1 = \{a, b, c\}$ and $S_2 = \{b, c, d\}$. Now, if $S_2$ is updated to $\{b, c', d'\}$ and synchronized with $S_1$, the update to $c \in S_1 \cap S_2$ should be propagated to $S_1$, which when synchronized becomes $\{a, b, c'\}$.

In the reviewed work, synchronization is usually not the main research topic. Typically the research concentrates on something that entails synchronization as a integral part, such as a distributed file system or a shared database. The only work that specifically deals with synchronization reviewed here is the SyncML synchronization protocol.

Although it would be interesting to review work that concentrates on synchronization, it is certainly beneficial to view it as an integral part of a system, as the described synchronization methods in these cases solve an actual synchronization problem. The holistic perspective also helps to recognize different aspects of the synchronization process, such as:

- Policies regarding when, what and how to synchronize.
- Conflict resolution mechanism, including conflict resolution policies.
- Low-level network transportation method (e.g. FTP, SSL).
- Caching and maintaining cache coherency.
- Consistency guarantees for synchronized data.
- Locking and session semantics.
- Methods for gathering and transporting updates (e.g. change log, deltas).

Throughout this document, we have tried to describe these aspects, to the extent they are applicable, for each of the reviewed systems.

Given the long-term goal of the MDIB, the review focuses on the synchronization mechanisms in distributed storage systems and how suitable these are for operation in a mobile environment. Particular points of interest on synchronization from a mobile perspective are:

- Are unexpected disconnections handled well?
- Does the protocol save bandwidth?
- Is the architecture suitable for peer-to-peer operation?
- Is the protocol or architecture too complex for mobile devices?
- Is the architecture scalable?
- Is the architecture secure?

The distributed storage systems included in the review are Coda, InterMezzo, OceanStore and Bayou. Coda and InterMezzo are distributed file systems, whereas OceanStore is a highly available and fault tolerant data storage facility designed to be deployed on a global scale. Bayou is a distributed database, designed from the ground up with disconnections in mind.

In addition, we review the SyncML synchronization protocol as an example of a synchronization protocol that is not tied to a particular application, and that is supported by a range of mobile devices. Furthermore, we take a look at the `rsync` protocol, which was specifically designed to enable efficient file updates over slow networks. Finally, we ponder the question of what "intelligent synchronization" could actually mean and give some examples, the most important of which is the 3DM tool for 3-way merging of XML data.

We are also interest in how easily a system could be utilized in the Fuego Core project. We have tried to evaluate each system in this respect as well, by looking as such issues as:

- Is the system well-documented?
- What is the API provided to application programmers?
- What is the implementation status?
- Is there any source code available?

## 4.2 Coda

The Coda file system [Bra98, SK92, MES95, Sat96, BBHS, S$^+$90, Cod], originating at the Carnegie Mellon University and building on the heritage of the famous Andrew File System, is perhaps the most well-known file system with support for weakly connected and disconnected operation. Since its initial development during the years 1990–91, and subsequent enhancement for weak connectivity in 1993–95 it has been extensively studied and refined. Coda is quite mature: the ongoing development concentrates on allowing Coda to be widely deployed.

The features of main interest in Coda are:

- Support for disconnected operation
- Support for weakly connected[*] operation with adaption to available bandwidth
- Free and relatively mature source code available
- Support for write-access to shared file systems in disconnected mode
- The ability to ensure the availability of important files during disconnected operation
- Cross-platform. Client and server software is available on several platforms[†].
- Excellent compatibility with legacy applications.

[*]High-latency, low-bandwidth (typically 9.6–64kbps) connections with occasional involuntary disconnections. Typically wireless links.

[†]They have even managed to get it running on Windows 9x.

In addition, Coda supports replicated file servers for higher performance and fault tolerance, server recovery, and client authentication and access control using a Kerberos-like scheme combined with access control lists.

The architecture of Coda follows the client/server paradigm. The design is optimized for access patterns exhibiting little to no concurrent writes to the same objects. On the server side, Coda stores files using its own scheme, meaning that you cannot just "start sharing" an existing file system.

Coda exhibits a hierarchical architecture. At the highest level of organization, there is the Coda *cell*. The clients and servers in a cell share the same common names pace and configuration information. Each cell has a server designated as the *System Control Machine* (SCM), which is responsible for the maintenance of the configuration databases. Movement of clients between cells is currently not possible, so a cell would typically contain the entire shared file tree of an organization.

The Coda name space, which appears as a directory hierarchy under a mount point (typically /coda) is populated by *volumes*. Volumes are subtrees of a server directory hierarchy, typically larger than a single directory but smaller than an entire partition. For instance, a home directory /home/ctl on server A could be exported to the Coda name space as /coda/home/ctl. You are allowed to export a volume inside the directory structure of another volume.

Volumes may be replicated across several servers for fault-tolerance. The group of servers replicating a volume is the *Volume Storage Group* (VSG), and the servers in the VSG available at a given instant to a client is referred to as the client's *Active VSG* (AVSG).

Security in Coda was designed on the basis that servers are trusted, whereas clients are not. For authentication and authorization during a session, Coda clients use a token obtained from a server in exchange for the correct password. Permissions are granted based on looking up the user who owns the token in the system's ACL.

### 4.2.1 Storage and Update Model

In this section, the operation of the Coda client and server is described first, after which the interaction between client and server is examined. Server-to-server communication is not described in detail, as the main interests lie in the mobile aspects of Coda.

On the client side, Coda makes aggressive use of caching, not only to improve performance, but mainly to make files available in states of weak connection or disconnection. The Coda client, consisting of a relatively small kernel-level module and the cache manager Venus, has three main responsibilities:

- Cache management: Checking currency of files in the cache, fetching files and making sure that the files the user has selected for off-line availability are in the cache.

- Propagating changes from the client file system to the server. Changes include modifications to files, directories and permissions.
- Maintaining a log of changes to the file system, when changes cannot be propagated immediately. This log is called the *client modification log*, CML

The server part of Coda, called Vice, performs the following tasks:

- Manages server storage. File data is stored on the file system provided by the OS. Coda metadata, such as volume and directory information, is stored in a transaction-enabled raw partition to provide fault-tolerance.[*]
- Grants and executes callbacks (in Coda terminology "breaks") to clients when an object is modified.
- Applies CMLs received from clients.
- Performs conflict detection and resolution.
- Handles server-to-sever replication, initiated by clients detecting stale data on a server.

Coda uses session semantics for shared files. The session starts when a file is opened and ends when the file is closed. Coda treats files as atomic objects, meaning that concurrent modifications at different locations in a file will result in conflicts. Consistency and recoverability of Coda metadata is provided through the use of the *recoverable virtual memory* (RVM) [MS91] transaction handling module.

### Hoarding, Emulating and Write-disconnected

The interaction between Venus and Vice takes place in three states: *hoarding*, *emulating* and *write-disconnected* (this state was called the *reintegrating* state before support for weak connectivity was added). These states correspond to being connected to a fast network (hoarding), disconnecting and working on the road (emulating) with occasional weak connectivity (write-disconnected) and finally returning back to the home network (hoarding). The states and possible transitions are depicted in figure 4.1.

In the hoarding state client changes to directories, files and permissions are immediately propagated to all servers in the AVSG (replication is thus

[*]Replaced by a binary file on Windows.

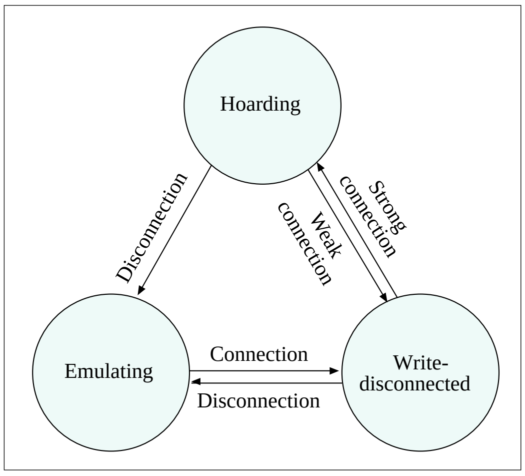


Figure 4.1: Venus states and state transitions.

primarily handled by clients sending their updates to all servers in the VSG) in addition to the locally cached copies. Each client chooses[*] a *primary server* in the VSG, from which it fetches current objects and receives callback notifications ("breaks") when objects on the server are updated. The callback notifications are thus used to mark cached objects invalid. A client alerts the primary server to initiate server-to-server replication, if it detects that any of the members in the AVSG has an old version of an object. Object currency is detected through the use of version vectors[†].

When Venus detects that the client has been disconnected from the network, it enters the emulating state. Ideally, the application user is not affected at all by the disconnection, as Venus tries to emulate connected operation in this state (hence the name of the state). The ability to continue using the file system as if nothing had happened during disconnection is a feature pioneered by Coda.

In disconnected operation, we can no longer be assured that the objects in cache are up-to-date, and the penalties for a cache miss are fatal — the file cannot be accessed at all. To prevent vital files from being absent from the cache, the user is able to specify a list of files, called the *hoard database,* that should always stay in the cache (so-called "sticky" entries)[‡]. As the modifications to the client cannot be propagated we store them in the CML, which is replayed on the servers in the AVSG once reconnected. As in the hoarding state, modifications are still applied to cached entries immediately. To save resources (and bandwidth at reconnection) the CML is subjected to optimizations, e.g. entries describing the creation, writing and subsequent

[*]Selection techniques include random selection or selection based on server load.
[†]See [P+83]
[‡]Files from the hoard database can still be evicted from the cache, if space is insufficient.

deletion of an object (the life cycle of a temporary file) are purged from the CML.

When the client is reconnected to the network (either over a weak or strong link), Venus enters the write-disconnected state, and the process of reintegrating the changes between the server and client starts. As modifications may have occurred on the server side, the cached entries on the client may be stale. Furthermore, to propagate the modifications local to the client we need to send the CML to the AVSG.

As the connection may be weak, we need to consider bandwidth usage during reconnection. Overlooking this fact made early incarnations of Coda unbearably slow when reconnecting over a weak link. Fortunately, with the introduction of rapid cache validation, trickle reintegration and user-assisted miss handling, performance over weak connections improved enormously. These techniques are described below.

To minimize cache validation traffic, version stamps for volumes as well as individual files are maintained. Version stamps for volumes enables validation of a large amount of files in a single sweep, provided no modifications have occurred to the volume, as is frequently the case. Using volume version stamps enables rapid cache validation.

Trickle reintegration is a process whereby the client continues generating updates to the CML instead of sending them directly to the AVSG, although the device is connected. The CML is allowed to age for some time, enabling optimizations to be done before it is sent to the AVSG. For instance, if the CML is allowed to age 10 minutes before being transmitted, we can optimize away a file create/delete pair 9 minutes apart. To increase responsiveness, an upper bound is set on the size of the transmitted CML chunks. The tradeoff of trickle reintegration versus hoarding is weaker consistency.

As opposed to fully disconnected operation, cache misses can be handled in the write-disconnected state. In case of a low bandwidth-connection, the delay experienced by the user when fetching large objects may, however, be prohibitive (a cache miss for a 1M file on a 9.6kbps link will cause a delay of some 20 minutes!). User assisted miss-handling means that in cases where huge transfer times[*] would result, Coda will query the user before initiating the transfer.

### Reintegration and Conflict Handling

Reintegration, which takes place constantly in the write-disconnected state, and when entering the hoarding state, reconciles the differences between a client and the servers. Reintegration consists of the following steps:

1. Venus reserves some resources from the server, which it has already tentatively handed out.

[*]Actually, a nifty mathematical model involving priorities and transfer times is used.

2. The CML is transmitted to the AVSG, which executes the CML operations, at the same time checking for conflicts. Some conflicts can be solved automatically (such as adding of files to the same directory), but not all: for instance, if a file has been modified on the server since disconnection, as well as on the disconnected client the conflict cannot be automatically solved. In such cases, it is possible to have Coda invoke *application-specific resolvers* to handle the conflict.

3. Updated files are fetched from the client.

If the CML causes a conflict, the log is rejected and the corresponding entries flushed from the client cache. In this case the user must resolve the conflicts manually.

### 4.2.2 Coda as a MDIB

When it comes to support for mobility, Coda is well though out. There is support for weakly connected operation, including unexpected and spurious disconnections. The weaknesses of Coda lie in the requirements not specifically related to mobility: it is not design to scale on a global to level, since there are no guarantees for strong consistency (ACID) nor support for transactions.

It should also be noted that the design of Coda, especially the authentication and authorization system, is based on the client/server model. An interesting question is if Coda could be extended to support operation in peer-to-peer mode.

### 4.2.3 Practical Issues

The sources for Coda are publicly available, and the system has reached a high level of maturity for research software. The semantics are easy to understand, lots of documentation is available, and the storage API is familiar to Unix programmers. In short, Coda should provide an excellent platform for experimentation.

## 4.3 InterMezzo

InterMezzo [Bra01, Bra02, IMW] is a newcomer in the family of distributed file systems with support for disconnected operation. The goal of the InterMezzo project is to achieve the same benefits as Coda as well as close to local file system performance, but with a simpler architecture.

The architecture of InterMezzo was heavily inspired by that of Coda and several researchers have been involved in both projects. Like Coda, InterMezzo makes use of aggressive caching and has session semantics based on file open and close. The update propagation scheme is similar to that of Coda's weakly connected operation. The main differences are that InterMezzo utilizes the underlying file system to a higher degree (no special partition required on the server) and that many performance optimizations unrelated to networking has been performed, such as moving code to kernel space and introducing asynchronous calls between the InterMezzo modules.

InterMezzo consists of two modules:

1. The kernel file system code, called Presto, which gathers a log of modifications to the file system. This log is called the *kernel modification log* (KML).
2. The cache manager, which is responsible for keeping the cache updated and sending the KML to the system's peer.

The cache manager originally consisted of a single program called Lento. Lento, which can still be used, has since been superseded by a HTTP-based approach consisting of a generic web server (such as Apache) and a program called InterSync. In the following discussion, we will assume that InterSync is used. Lento essentially does the same as InterSync combined with a web server, differences are mainly in how communication is initiated. The use of a web server for communications automatically adds support for secure transfers and proxies to InterSync.

### 4.3.1 Storage and Update Model

InterMezzo is a filtering file system [HP94] that sits on top of an existing *journaling* file system capable of supporting nested transactions, such as Linux' ext3, ReiserFS and XFS. The underlying file system stores files in the same directory hierarchy and with the same names as those seen in InterMezzo file system, with the addition of some special files and directories for control purposes. InterMezzo relies on the journaling abilities of the underlying file system to handle recovery and guarantee consistency. The advantage of this design is that it leverages the performance and transaction handling capabilities of an existing file system (unlike Coda which uses its own transaction handling system, the RVM).

InterMezzo can be set up for both one- and two-way synchronization. In the former case, changes are only propagated from the server to the client. In this case it is sufficient for the client to use InterSync without having an InterMezzo partition, since no modification log needs to be sent to the server.

In the case of two-way synchronization, we need to track changes on the client as well, and thus the shared files must reside on an InterMezzo partition. Two-way synchronization can be performed in two different

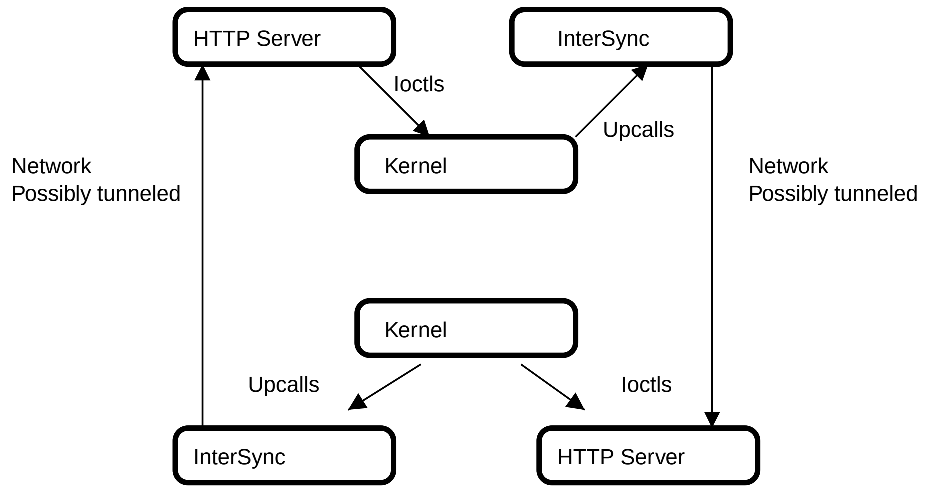


Figure 4.2: Symmetric two-synchronization operation in InterMezzo [Bra02].

ways: one is to use a web server on the server side only, in which case the client KML is pushed to the web server, the other is to have web servers on both client and server. In the latter case, which we will examine, the client and server operate symmetrically. See figure 4.2.

Modifications on the server side are propagated to the client by having InterSync fetch the KML (through a normal HTTP file request) regularly from the server. InterSync then processes the KML, fetching the corresponding file from the server whenever a file modification record is encountered. As an alternative to polling the KML, synchronization may also be initiated by a message from the server whenever modifications occur. Furthermore, InterSync can be configured to only fetch the modified files when they are actually accessed on the client. Some optimizations are done when processing the KML, such as not fetching temporary files or fetching the same file multiple times.

Modifications on the client are propagated similarly by having the server download the KML of the client.

The scheme described above raises the question of how a server handles the KML when synchronizing to several clients (with different times of previous synchronization). The answer is that the server stores the last successfully retrieved record of the KML for each client, and only sends more recent records of the KML to the corresponding client. Unbounded growth of the KML is prevented by having it periodically truncated. To be able to restore a heavily out-of-date client, InterMezzo maintains another log, the *synchronization modification log* (SML), which only contains object creation records.

Presumably, one can use server-granted write permits (callbacks), such as those used in Coda, to guarantee a higher degree of consistency. Unfortunately, permit handling is described too vaguely in documentation to give an overview here.

Conflict detection is handled by checking the KMLs for conflicting operations. Assume that the KMLs gathered since the point of synchronized directory trees are $L_1$ on the client and $L_2$ on the server. The client receives $L_2$ in order to perform reintegration. We need to check for possible conflicts between the logs $L_1$ and $L_2$, such as modifications to the same file.

Conflicts are automatically resolved according to a given policy by generating new logs $L_1^{local}$, $L_2^{local}$ and $L_1^{remote}$, so that $L_1^{local}L_2^{local}$ when applied on the client, yields the same result as applying $L_2L_1^{remote}$ on the server ($L_1^{remote}$ is the KML sent to the server). The following policies are mentioned:

- Mobile policy and High availability policy*. The conflicting is kept on the client. When a conflicting server object is detected, the client object is moved away to another location.
- Re-synchronization policy. Used for heavily out-of-date systems, which need to apply the SML.

The details of generating $L_1^{local}$, $L_2^{local}$ and $L_1^{remote}$ can to some extent be found in [Bra02]. The need for $L_2^{local}$ and a $L_1^{local}$ differing from $L_1^{remote}$ arise due to the fact that conflict resolution mechanism may require different operations to take place on the server and client.

### 4.3.2 InterMezzo as a MDIB

InterMezzo implements basic support for disconnected operation, and can thus operate in a mobile environment. However, there is no explicit support for weakly connected operation as there is in Coda. Noting the large improvement in performance that was achieved in Coda by accounting for weak connections, one suspects that InterMezzo could be improved as well. Techniques that come to mind is protocol optimization and delta transfers.

InterMezzo was designed for simplicity, which should be advantageous in mobile environments. Especially the layered file system design and KML gathering seem efficient and elegant, provided that an underlying journaling file system is available. On Linux-based platforms, this should not be a problem; there is even a journaling file system for flash devices available: JFFS2 [Woo].

High availability and scalability have been considered to some extent, but cannot be compared to the massive approaches taken in e.g. OceanStore (see the next section).

*The exact difference between these remains elusive in [Bra02]. In fact, it is stated that they are the same if “the failed node [in a fail-over cluster] is the client”

Although the basic design of InterMezzo is client/server, it exhibits a great deal of symmetry between these. This speaks for easy adaption to peer-to-peer operation. Compared to Coda, the lack of a special file structure on the server should be advantageous and enable users to start sharing any file system almost “ad-hoc”.

### 4.3.3 Practical Issues

Sources and documentation for the InterMezzo project are available from the project web site. There is some testimony that InterMezzo has reached a level of maturity where it can be used on a day-to-day basis [Bar02]. The sharing semantics are familiar from Coda and the API is the standard Unix file API.

On the downside, we note that there are only a few documents on InterMezzo, some of which appear outdated ([Bra01]) and others rather unfinished and unclear ([Bra02]). “Use the source, Luke[*]” appears to be an appropriate motto for those wishing to learn the details of InterMezzo. The use of kernel code (especially with the introduction of InterSync) will make deploying on other platforms than Linux harder.

Security in InterMezzo seems to be fairly rudimentary as it is of now.

## 4.4 OceanStore

The OceanStore project [$R^+$01b, $K^+$00, OSW] at University of California, Berkeley is an attempt at constructing a secure highly available and reliable storage system on a global scale. The system is envisaged to support the storage needs of some 10 billion users, amounting to a total capacity of roughly $10^{18}$ bytes. The distinguishing features of OceanStore is its massive scale, content-based routing, strong support for security and use of introspective techniques to optimize the performance of the system.

The fundamental unit of storage in OceanStore is an encrypted binary object, identified by a fixed-length globally unique identifier (GUID). Objects are stored persistently and new versions are automatically created with each update. To give the illusion of mutable objects, there is a naming mechanism by which the latest, or *active*, version of an object can be addressed.

To facilitate fault tolerance and increase performance, active objects are automatically replicated and distributed among the network nodes as seen fit by the system. Older versions of an object are stored in a highly fault-tolerant *archival* mode. In archival mode, an object is encoded using erasure codes and divided into fragments, which are spread over a large number

[*]That is: read the source code.

of servers. To reconstruct the original object, any sufficiently large subset (e.g. 30%) of the fragments may be used.

The devices participating in the OceanStore infrastructure are nodes in an overlay network[*] on top of an existing IP infrastructure. The main feature of the overlay network is its ability to route messages (e.g. data reads) to the the closest[†] instance of a stored object. As this routing mechanism is one of the fundamental enablers of fault tolerance and replication in OceanStore, we will look at it in some detail in the next section.

The architects of OceanStore have done their outmost to eliminate single points of failure from the system. This is reflected throughout the design, and especially in the subsystem called the *inner ring* (or *primary tier*[‡]), which handles global ordering and commitment of update operations as well as provides a source of data with strong consistency guarantees. The task that the inner ring performs is typically handled by a single authoritative server in other distributed storage systems (e.g. Bayou and Coda).

Each object is assigned a set of servers, which form its inner ring. The servers in the inner ring agree on updates using a Byzantine agreement protocol [LSP82], guaranteeing consistency among the replicas. The protocol allows any $m$ out of $3m + 1$ servers to fail, without the ring going inoperational. The penalty for this high level of fault-tolerance is a large amount of network traffic between the servers in the inner ring[§].

In addition to the replicas in the inner ring, there may exist *secondary replicas* of an object throughout the system, distributed in the form of trees rooted at servers in the inner ring. The secondary replicas do not provide any guarantees regarding consistency or currency.

Updates are propagated to the secondary replicas in three manners: they are propagated down the tree of replicas from the inner ring, replicas may send requests for updates up the tree, and finally in an *epidemic* manner (as in Bayou). During epidemic update, the replicas quickly spread tentative updates among themselves and pick a tentative serialization order. Tentatively updated replicas can be read by applications which do not require strong consistency guarantees. See figure 4.3.

To the application developer OceanStore provides its services through sessions. Sessions may be created with different levels of consistency guarantees, similarly to Bayou. As expected, there is a tradeoff between consistency guarantees and connectivity: in case of disconnection or weak connectivity, strong consistency cannot be guaranteed.

The basic assumption regarding security in OceanStore is that the in-

[*]That is, an application-level virtual network on top of an existing physical one, such as Gnutella.

[†]According to some metric, e.g. latency.

[‡]The terminology varies: [K$^+$00] uses *primary tier*, [R$^+$01b] uses *inner ring.*

[§]It should be noted that in [K$^+$00] it is argued that the overhead of the Byzantine agreement protocol is not in fact very large.

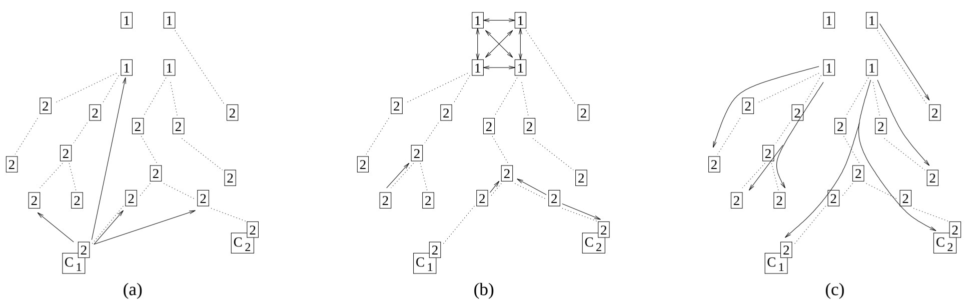


Figure 4.3: The path of an update (figure and text from [K+00]). (a) After generating an update, a client sends it directly to the object's inner ring, as well as to several other random replicas for that object. (b) While the inner ring performs a Byzantine agreement protocol to commit the update, the secondary replicas propagate the update among themselves epidemically. (c) Once the primary tier has finished its agreement protocol, the result of the update is multicast down the multicast tree to all the secondary replicas.

frastructure is untrusted. In practice this means that no node besides the client is allowed to see unencrypted data, limiting read access to those in possession of the encryption key. Write access is controlled through signed access control lists. By being very fault-tolerant OceanStore should also be relatively immune to denial-of-service attacks. Measures have also been taken to prevent malicious servers from replacing or changing objects they do not have access rights to. However, we cannot get away without trusting anyone: there are some specially assigned servers that we will need to trust to carry out protocols as well as distributed consistency management for us.

OceanStore was designed to be self-maintaining. This implies two fundamental properties: fault-tolerance (the mechanisms of which were described above) and self-repair. The mechanisms for self-repair in OceanStore include the ability to automatically handle both unexpected and advertised insertions and removals of nodes, processes that monitors the network for suboptimal routes and that periodically sweep through the OceanStore to check and repair objects (using archival fragments) as well as models for predicting server and disk failures.

To further reduce the need for manual tuning, *introspective* processes,i.e. processes that observe the system and make tunings based on the observations, have been deployed. Introspection is used for cluster recognition, whereby clusters of closely related files (e.g. a set of files a user is actively working on) are recognized. This clustering information, along with information on usage patterns, is utilized by another introspective process, the replica manager. The replica manager is able to intelligently prefetch files to a server close to the user: your mail is automatically fetched to your workstation during office hours and to your PDA while traveling. If this actually works in practice remains to be seen; the authors appear optimistic, nonetheless.

### 4.4.1 Routing to Data

According to [K+00] OceanStore uses a twofold approach for routing to data. First, a fast probabilistic algorithm, based on Bloom filters, is used to look for the data in nearby nodes. Bloom filters [Blo70] are a way of compactly representing sets, with some probability for false matches. The nodes store Bloom filters for the objects in adjacent nodes up to some depth. If a filter indicates a match we can route directly to the corresponding node. The probabilistic routing algorithm is not mentioned in the later publication [R+01b], indicating that it might have been dropped.

If the probabilistic routing fails, a slower, reliable method based on the Tapestry [ZKJ01] overlay and routing infrastructure, also developed at UC Berkeley, is used for routing the messages to the closest instance of the stored object. Tapestry implements a variation of the routing mechanism introduced by Plaxton, Rajaman and Richa in [PRR97], with enhancements for fault-tolerance and dynamic insertions and removals of nodes. The routing and location method used in Tapestry uses the same general method, described below, as that of Plaxton et al.

Assume a name space of $b^k$ nodes. Each node $N$ in the network is assigned a unique address in the range $[0, b^k - 1]$. When routing to a destination node $D$, the address of the node (which we will also denote with $D$, since there is no risk of confusion) is divided into $k$ blocks $b_{k-1} \dots b_0$ so that $D = \sum_{i=0}^{k-1} b_i b^i$, with each block in the range $[0, b-1]$. For instance, using $b = 16$ and $k = 4$ the blocks of the address AB09 (base 16) would be 9, 0, B and A.

Each node $N$ maintains $k$ neighbor maps with $b$ entries, where the entry $(i_k, i_b)$ contains the route to the closest node $C$ (including $N$ itself), whose $i_k$:th block equals $i_b$ and the blocks $i_k - 1 \dots 0$ match the corresponding block in $N$.

Routing to a destination node $D$ is done one block at a time, starting from block 0. Assume $i$ is the current block. The next hop from a node $N$ is given by looking up the node at position $b_i$ in the $i$:th neighbor map, i.e. the entry $(i, b_i)$. In this way, the destination is solved digit by digit. The maximum length of the path is trivially $k$ (but usually less, since several neighbor map entries are loopback entries). For a routing example, see figure 4.4.

Locating an object $O_1$, located at a sever node $S$ works as follows. A hash function is used to calculate a mapping from $O_1$ to a node $R$, which is the "root node" of the object. When $O_1$ is created at $S$ it is published to the infrastructure by sending a message $\langle S, O_1 \rangle$ from $S$ to $R$, which states that

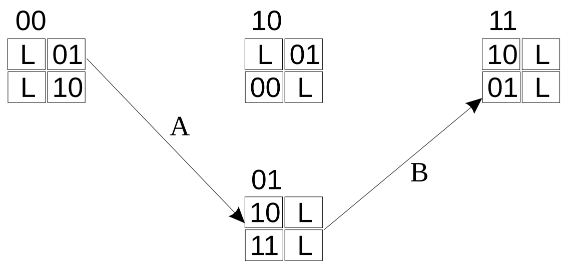


Figure 4.4: Routing using the algorithm of Plaxton et al. In this example $b = 2$, $k = 2$ and the distance measure is the geometrical distance between the nodes in the figure. The address of the node is written over the node's routing table. In the routing table, the entry $(i_k, i_b)$ is at row $i_k$ column $i_b$; $L$ means that the entry points back to the node (a loopback entry). The arrows $A$ and $B$ show the routing of a message from node 00 destined for node 11.

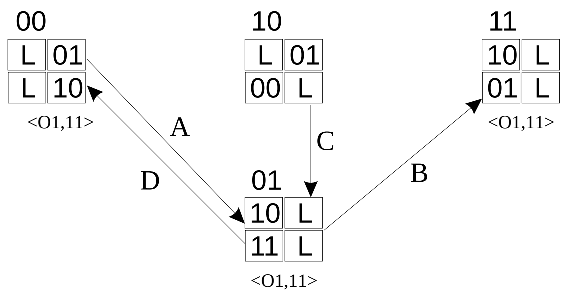


Figure 4.5: Object lookup using the algorithm of Plaxton et al. The node 00 publishes $O_1$, whose root node is 11. The arrows $A$ and $B$ indicate the path of the publishing message. When 10 queries for $O_1$, it sends the query to 11. However, the location of $O_1$ is discovered at 01 after the first hop, and the message is sent directly to node 00 (arrows $C$ and $D$).

$O_1$ can be found at $S$. All nodes on the path from $S$ to $R$ (including $R$) store this information. When addressing $O_1$ from a node $C$ we send a message destined for $R$ ($R$ is obtained from $O_1$ using the same hash function that $S$ used). At some point (at latest when reaching $R$) the message will route through a node that has knowledge of $O_1$, which then is able to forward the message to $S$. See figure 4.5. If a replica of $O_1$ is published at another server $S'$, the publishing works similarly, with the addition that the message $\langle S', O_1 \rangle$ replaces $\langle S, O_1 \rangle$ at each node where $S'$ is closer than $S$.

### 4.4.2 Update Model

OceanStore provides different guarantees on consistency and data currency. Locks are not used, avoiding the problems traditionally associated with locking, e.g. stale locks and too aggressive locking preventing sharing of data. The basic ideas used are:

- An update consists of (*predicate*, *action*) pairs. The actions of the first predicate that evaluates to true is executed atomically, and the update commits. If no true predicate is found, the update aborts.
- A strongly consistent and current copy of an object can always be obtained from the servers in the inner ring.

These ideas are similar to those used in the Bayou system, with the difference that merge procedures are not used, and the inner ring has the role of the primary replica in Bayou. The fact that only ciphertext is stored in OceanStore limits the available predicates that can be used in the update operations, and the substitution of a primary replica with a group of server complicates the update procedure. In [$K^+00$] the authors acknowledge that these issues are problematic, and not yet resolved.

The predicates available for updates are *block-compare*, predicate on metadata, such as *compare-size* and *compare-version*, as well as a *search* predicate for searching the ciphertext for an string (without revealing the cleartext of the search string). Assuming certain types of block cipher, such operations as *replace-block*, *delete-block* and *insert-block* are available.

ACID semantics can be achieved by using the *compare-block* predicate to check the read set of the transaction, and write the updated data with *replace-block*. If an application requires that reads returns data that is current and strongly consistent (e.g. a banking transaction) it needs to communicate with the inner tier; as the secondary replicas may contain outdated and tentative data. Strong consistency is thus not available to a mobile device (unless it itself is the inner tier).

One may question whether the update scheme really is sufficient for handling concurrency. Consider the transaction $\{a = a + 100, b = b - 100\}$ (which may signify the transfer of \$100 from account $b$ to account $a$). In OceanStore, we need to read the amounts $a$ and $b$, and then construct an update that checks that the amounts are the same that we read before updating them. The problem is that we cannot do the read and update atomically, and thus there is always the possibility that another transaction updates the amounts in between — in which case the update aborts and we must start all over. However, there is no guarantee that the update will succeed on the second try, or any subsequent for that matter.

The predicate-action update scheme is also able to handle simple conflict resolution automatically. As an example, if we always want the more recent version to persist, we can use the *compare-version* predicate to check if the version number has increased since the last read, and discard the write if that is the case.

The mapping between object names (such as a human-readable file name) and object handles (which are used internally to access objects, i.e. GUIDs in the case of OceanStore) are handled by hashing over the object name and some additional information, such as an encryption key. Objects can also be named by hashing over the object's content, as in the case of archived objects.

### 4.4.3 Mobility and OceanStore

The authors speak of the possibility for disconnected operation in OceanStore. And indeed the fundamental enablers are there: local replicas can be stored in the mobile device, and data writes need not propagate immediately. There are, however, several aspects of OceanStore which are suboptimal in the mobile environment under consideration in Fuego Core.

There is no mentioning of the protocols used by OceanStore having been optimized for saving bandwidth or roundtrips. The amount of different protocols active in OceanStore appears quite large: client update requests, epidemically propagated updates, pushed updates to secondary replicas, update requests from replicas, several introspective processes exchanging system state as well as processes sweeping through all the stored objects. In a mobile environment, presumably several of these would processes need to be disabled to conserver bandwidth.

A general design principle of OceanStore seems to use soft state whenever possible. Soft state means that the state information decays over time; if not refreshed frequently enough the state is deleted. The requirement for refreshes at certain intervals does not fit well into the mobile world.

Client updates are not optimized for limited bandwidth, there are for instance no delta transfers; furthermore, the use of ciphertext makes it harder to add such transfers to the protocol.

Putting a member of the inner ring on a mobile device seems quite infeasible, given the additional overhead of the rather heavy Byzantine agreement protocol (an estimated 6 roundtrips and $O(n^2)$ amount of communications [K+00]). In practice this means that no mobile device can contain an authoritative replica of an object.

Unexpected disconnections is another issue that the system needs to deal with. Practical implications are that protocols should minimize the amount of state and connection buildups and teardowns should be light. The update messages seem to be stateless and hence well suited for mobile operation. Connection buildups, on the other hand, may be problematic, as a Tapestry node upon entry in the network needs to build a routing table. The routing table can be cached, but if one moves to a different access point, the old routing table may not be valid. Routing over the wireless hop should probably not be done using Tapestry.

Judging from [R+01b, K+00] OceanStore uses many rather complex protocols, and is itself complex. However, in a mobile environment, we need to minimize complexity as CPU cycles and bandwidth are usually very constrained. As for security, the fact that OceanStore was designed with strong security in mind from the ground up is a merit in the wireless environment. Furthermore, OceanStore is not dependent on a server infrastructure, and should thus be able to work in peer-to-peer mode.

### 4.4.4 Practical Issues

So how does one test and deploy OceanStore? The short answer is: you don't. At the time of writing OceanStore is not available for download. [R+01b] states that the system is not yet "fully operational". However, the underlying routing framework, Tapestry, is available for download at the OceanStore website [OSW].

To the programmer, OceanStore provides (or is envisaged to provide) several alternatives. The base API provides full access to OceanStore functionality in terms of sessions and session guarantees, updates and callbacks (which are used to inform the application of when e.g. an update commits or aborts). On top of this API, more familiar APIs (called *facades*) have been implemented: a Unix file system, a transactional system, and a WWW gateway. The facade APIs allow legacy applications to harness the benefits of OceanStore.

## 4.5 Bayou

Bayou [PST+97, E+97, D+94, BaW] is a storage system designed with the collaboration of frequently disconnected users in mind. In Bayou, the database used by a collaborative application is aggressively replicated to provide high availability. To facilitate work in off-line mode Bayou abandons the requirement for strong consistency among replicas and instead introduces a number of different guarantees for weakly consistent operation.

New data may be written to any available replica. From there changes will eventually propagate to all replicas by means of the update propagation mechanism, which in Bayou is *anti-entropy*. A very interesting idea in Bayou is the bundling of write checks and merge procedures with database updates.

Other important features of Bayou are:

- Use of committed and tentative data (as in OceanStore).
- Different session guarantees for weakly consistent data are provided.
- Built on the relational database model.

- Should be suitable for peer-to-peer operation due to the anti-entropy update mechanism and easy replica insertion and deletion.

The project was carried out during the years 1993–97 at the famous Xerox Palo Alto Research Center (PARC). During the course of the research project several collaborative applications were built on top of Bayou, including a group calendar and an email application.

### 4.5.1 Storage and Update Model

The storage system at each replica consists of a log of writes[*] and the database that results from applying these writes in order. In theory, the write log on each server contains all writes to the database,[†] received either from clients or other replicas. The task of the update mechanism is to reach an eventual agreement among all the servers on the set of writes in the log, as well as the order of the writes.

When a server receives and accepts a *client write* it assigns the write an *accept stamp*, and associates the server ID with the accept stamp. The accept stamps are assigned in a monotonically increasing fashion, and define an ordering for all the client writes received by a specific server: if write A is accepted before write B, A will precede B in the ordering. This ordering is called the *accept-ordering*. The accept ordering is maintained in the write log.

When receiving a write from another server, the write already has an accept stamp, which is left unmodified. Each server stores the last received accept-stamp in a version vector, indexed by the server that assigned the accept stamp. For instance, if the version vector on a server $S$ were $\{S : 1000, S_1 : 2002, S_2 : 3003\}$, it would mean that the accept stamp of latest client write on $S$ was 1000, and that it so far has received client writes to $S_1$ up to accept stamp of 2002, and client writes to $S_2$ up to 3003.

Using these concepts, we can present the basic operation of the anti-entropy update propagation mechanism:

Write operations are propagated between pairs of servers. A server $S$ propagates the writes it has received by contacting a randomly chosen replica $R$, and asks $R$ for its version vector $V$. $S$ then iterates through its write log, sending any write from a server $s$ with an accept stamp $a$ to $R$, if $V[s] < a$. That is, $S$ only sends writes to $R$ that $R$ has not seen previously. The pseudocode for basic anti-entropy is shown in figure 4.6. Using this scheme, all writes will eventually reach all replicas, according to the theory of epidemics. Also note that the accept ordering allows compact repre-

[*]A *write* in Bayou terminology is a procedure that generates a set of updates to be applied to the database. A write may for instance delete a row in a relational table.

[†]In practice, writes that have been committed can be discarded from the write log.

```
anti-entropy(S,R) {
    Get R.V from receiving server R
    # now send all the writes unknown to R
    w = first write in S.write-log
    WHILE (w) DO
        IF R.V(w.server-id) < w.accept-stamp THEN
            # w is new for R
            SendWrite(R, w)
        w = next write in S.write-log
    END
}
```

Figure 4.6: Basic anti-entropy executed at server *S* to update receiving server *R* [PST+97].

sentation, in the form of version vectors, of all the set of writes seen by a server.

In practice Bayou exhibits some modifications to the basic anti-entropy scheme:

- Writes are divided into *committed* and *tentative* writes. One database replica is designated to be the *primary replica* and it determines a total ordering of writes by assigning a monotonically increasing *commit sequence numbers* (CSNs) to all writes it receives.

  On all servers, the writes that have been assigned a CSN are committed. The committed writes are ordered before any write without a CSN in the write log. Writes without CSNs are tentative writes.

  To accommodate for this modification, each server *R* maintains a counter of the highest CSN assigned to a write. During anti-entropy, the sender *S* checks if it has a higher CSN, and in that case sends the CSNs of all writes between *R.CSN* and *S.CSN* to *R*, along with any writes unseen to *R*. *R.CSN* is then updated to *S.CSN*.

- The introduction of committed writes allows Bayou servers to truncate the write log. Any write with a CSN may be removed from the write log, as we know that its position in the log will no longer change and hence its effect on the content of the database has been determined. The tradeoff is that if we need to perform anti-entropy with a server beyond the log truncation point, the entire database needs to be transmitted.

- Anti-entropy through transportable media. We can easily export the update log from a given starting point to e.g. a CD-ROM, and play back this log on a number of replicas. The receiving replicas just ignore any updates they have seen before.

```
Bayou_Write(
   update = {insert, Meetings, 12/18/95, 10:00am, 60min, "Project Meeting: Kevin"},
   dependency_check = {
      query = "SELECT key FROM Meetings WHERE day = 12/18/95
         AND start < 11:00am AND end > 10:00am",
      expected_result = EMPTY},
   mergeproc = {
      alternates = {12/18/95, 12:00pm};
      newupdate = {};
      FOREACH a IN alternates {
         # check if there would be a conflict
         IF (NOT EMPTY (
            SELECT key FROM Meetings WHERE day = a.date
             AND start < a.time + 60min AND end > a.time))
               CONTINUE;
         # no conflict, can schedule meeting at that time
         newupdate = {insert, Meetings, a.date, a.time, 60min, "Project Meeting: Kevin"};
         BREAK;
      }
      IF (newupdate = {})   # no alternate is acceptable
         newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, "Project Meeting: Kevin"};
      RETURN newupdate;}
)
```

Figure 4.7: A bayou write for a group calendar [E+97].

- Casual ordering of writes. To provide the session guarantees presented later on, a casual write order is introduced, which allows us to determine if at the time of write $B$ to a server $S$, another write $A$ was known to $S$. This can be implemented with a modification to accept stamp assignment which still preserves the condition of monotonic increase.

- Light-weight server creation and retirement. Bayou use special writes (that propagate as normal writes through anti-entropy) to indicate server creation and retirement. To accommodate for light-weight server creation and retirement, each server needs to support version vectors whose size can be dynamically adjusted.

The write operations in Bayou were designed to be very flexible to account for the loss of strong consistency. Each write operation has three parts: a dependency check, an update set and a merge procedure.

The update set consists of updates, insertions and deletions to the database. The dependency check specifies the conditions that must hold in order to apply the update set to the replica. It consists of a generic SQL query and the expected result of the query. A write passes the dependency check if the query returns the expected result when executed on the replica. The dependency check is thus used to detect conflict situations.

If the dependency check fails, the merge procedure part of the write is executed. The purpose of the merge procedure is to provide alternate courses of action when the update set could not be applied. In other words, the merge procedure can be used to handle conflict resolution, but may also defer it by e.g. writing the offending record to an error log.

As an example, consider the write to a group calendar in figure 4.7. The update set of the write tries to reserve the conference room for Kevin from

10 to 11am on December 18th. The dependency check of this write is to verify that the conference room is indeed empty at this time. The merge procedure tries to reserve the conference room at an alternate time, and if that fails, notifies Kevin by inserting the failed reservation in an error log.

[E+97, D+94] present a number of session guarantees for weakly consistent replicated data that have been implemented in the Bayou project. These are:

**Read Your Writes** ensures that reads within a session sees any previous writes within that session. Without the RYW guarantee, a deleted message could reappear during a mail reading session.

**Monotonic Reads** guarantees that successive reads will see an increasingly current view of the database. That is, if an application has seen the effects of a set $W$ of writes, the set of writes seen by subsequent reads, $W_1$, are guaranteed to be larger than $W$: $W_1 \supseteq W$. Without the monotonic reads guarantee, you might see a document in a directory listing, but get a "document does not exist" error when opening it.

**Writes Follow Reads** ensures that traditional write/read dependencies are preserved in the ordering of writes at all servers (i.e the guarantee holds across sessions). WFR entails constraints on write operations with respect to ordering and propagation; if one of these is relaxed we get the WFR Ordering and WFR Propagation guarantees. Without the WFR guarantee, you could see responses to an article in a newsgroup before seeing the original article.

**Monotonic Writes** states that within a session writes must follow previous writes. As an example need for monotonic writes, consider storing a library and subsequently the application that uses it in a Bayou database. Without monotonic writes, you might see the application, but not the library.

### 4.5.2 Mobility Issues

Bayou was designed with disconnected operation in mind, and hence works quite well in that respect. Anti-entropy handles disconnections well, as the process can simply continue from the last received write when the connection is re-established. Furthermore, only a small amount of state is needed before the transmission of writes can start. No optimizations have, however, been made for weakly connected operation. As stated in [PST+97], quite a lot of bandwidth could be saved by optimizing the write messages propagated during anti-entropy.

The fact that each mobile device may work as a server raises some serious concerns regarding security, as one cannot naively trust every mobile server to be friendly. A discussed in [S+97], some level of security can be achieved by digitally signing each write operation and maintaining a trusted server site that keeps a full log of operations. The overhead of essentially having to sign every message is, however, not negligible and will impact performance.

Although the basic idea of anti-entropy update propagation appears simple, and thus should be suitable for implementation on mobile devices, one should not overlook that the current design requires an underlying database as well as an interpreter for the merge procedures.

### 4.5.3 Deployment Issues

The only documentation available on Bayou appears to be the papers listed at the project's web site. No source code nor binaries appear to be available for download. Deploying a Bayou-like system would most likely involve rewriting our own implementation from scratch.

## 4.6 SyncML

SyncML [Pab02, Syn02b, Syn02a, SyW] is an industry initiative to standardize the way data synchronization is handled in mobile devices. Synchronization has traditionally been handled in an application-specific and often proprietary manner, which leads to limited interoperability between applications, lack of support for different transport methods as well as an inconsistent user experience, due to differing designs. The goal of the initiative is to be able to remedy this situation, and enable "mobile devices that support synchronization with any networked data" (and vice versa). Participating companies include wireless heavyweights Ericsson, Nokia and Motorola as well as IBM.

To this end the SyncML initiative has specified a synchronization framework. The essential parts of the framework are the SyncML Synchronization protocol, the SyncML Representation Protocol as well as bindings for various transport protocols, such as HTTP and OBEX. The synchronization protocol defines the high-level interaction between a device and its peer. This entails connection setup, authentication, synchronization (in several modes) and object ID mapping procedures. The representation protocol presents the protocol messages in detail in terms of syntax, parameters and result codes. Consider authentication for instance: the former specification defines when and which messages are sent, while the latter gives the format for each of the messages sent. The representation protocol also introduces the ability to filter and search the database as well as execute commands on the peer.

For the reminder of this document, we will simply refer to the various protocol specifications as the SyncML protocol. The SyncML protocol has the following features:

- Multiple transport protocols. SyncML can work over HTTP/TCP, OBEX, WSP and Bluetooth.
- Support for synchronization of any data that can be expressed as a collection of (*key,value*) pairs. Values may be arbitrary binary data, including XML documents, vCards, email messages etc.
- Optimized for the mobile environment.[*]
- Leverages existing technologies such as XML, HTTP and TCP/IP.

The SyncML architecture is client/server, the mobile device normally being the client and its strongly connected peer the server. The synchronization mechanism is based on the transmission of updates between the client and the server. Typically, the client sends its updates to the server, which reintegrates them (possibly solving conflicts). The server then sends its set of modifications (including possibly resolved conflict entries) back to the client, which stores them in its database. The update operations allow insertion, deletion, replacement and copying of objects.[†] Several modes of synchronization are supported, used to bring either or both devices up to date, and accounting for the possibility of a missing update log:

- Two-way synchronization. Client-initiated synchronization, where updates are transferred from client to server and vice versa.
- Slow sync. The client transmits its entire dataset to the server, after which the server transmits updates to the client. Used instead of two-way synchronization if the client update log cannot be used for some reason (e.g. the log is lost).
- One-way sync from client. Updates are transferred from client to server only.
- One-way sync from server. Updates are transferred from server to client only.
- Refresh sync from client. The client sends its entire dataset to the server, overwriting any corresponding data in the server database.
- Refresh sync from server. Like refresh from client, but the entire server dataset is transferred to the client.

[*]It would indeed be very interesting to know exactly how SyncML was optimized for the mobile environment. Besides a few spurious comments in the specifications, the author has found no document justifying this claim.

[†]A somewhat odd limitation is that there is no operation for renaming objects, presumably due to the way IDs are assigned in SyncML.

The SyncML protocol does not specify how conflicts are resolved; this matter is up to the implementation of the synchronization engine. The protocol does, however, define some messages and status codes relating to conflict resolution, e.g. a status code indicating that a conflict was resolved by merging the conflicting records.

To know from which point in the update logs synchronization should occur, as well as to enable devices to synchronize with multiple peers, synchronization *anchors* are used. The anchors mark positions in the update logs (similarly as positions in the KML in InterMezzo are remembered) for each peer, and are used to check that no updates are lost or being applied more than once on successive synchronizations.

When adding items in disconnected mode, there is the question of how to allocate new object IDs. In systems such as Coda, this is solved by tentatively assigning IDs, which are then verified upon synchronization. SyncML takes a different approach: each client manages its own set of IDs (called local IDs). During synchronization a map (stored on the server) between local IDs and server (global) IDs is generated.

The authentication methods supported by SyncML works similarly to HTTP basic and digest authentication.

### 4.6.1 Deployment Issues

At a first look, SyncML seems like a good protocol to use when providing a synchronization interface to existing software or to use as a synchronization platform on top of which more advanced concepts, such as intelligent synchronization policies, can be built. Upon closer inspection, however, one cannot help noticing some issues that will make it harder to use SyncML in practice:

- The SyncML Reference Toolkit is no longer available for download (it has been at some point). Scary-looking legalese at the site warns you that the SyncML protocols may be covered by patents.

- The SyncML web site is lacking usable information for developers. The only thing you will find are the protocol specifications.

- The specifications are quite confusing and badly organized; finding information is hard. [Syn02b], for instance, references non-existing chapters in representation protocol documents.

- [Ste01] concludes that the industry (excluding the founders of SyncML) have been slow on adopting the protocol.

Furthermore, the SyncML protocol is not very simple, given the relatively straightforward task it sets out to provide a solution for. One may ask what a database search operation has to do in the protocol, as it adds a great deal of complexity by requiring that the protocol is able to access the data it synchronizes, not just to transport it.

## 4.7 Adding Intelligence

In the Fuego Core project we want to go beyond plain data synchronization by adding "intelligence" to the synchronization mechanisms. The term *intelligent synchronization* does not appear* to be established in the scientific community, and thus we need to define what it actually means. The current thinking in the project is that intelligent synchronization would entail:

- Policies.
  - when and where to synchronize (depending on connectivity, pricing, demand etc.).
  - what to synchronize (automatically fetch files that are important in the current user context).
  - how to synchronize (different methods yield different tradeoffs between bandwidth and CPU cycles).
- Synchronization across "similar".† data.
  - synchrnization across different data formats.
  - synchronization in the form of 3-way merging.

### 4.7.1 Synchronization Policies

Examples of synchronization policies that address different aspects of when, what and how can be found from existing systems. Some examples are listed below:

**Coda** In [MES95] a model of user patience is used to determine if a cache miss should be handled transparently or if the user should be asked for confirmation before fetching the object. The threshold depends on available bandwidth and the priority of the object (which the user can set). For instance, a 1 megabyte object at medium priority would be fetched transparently at a link speed of 64kbps, whereas the user would be asked for confirmation on a 9.6kbps link.

*This claim is based on web searches and searches in some publication databases.

†This notion has not yet been formally defined, but presumably would go along the lines that two data sets $A$ and $B$ are similar if there exists a function $f$ that is one-to-one and onto such that $f(A) = B$.

[HKZ02] describes an extension to Coda that allows users to either specify the maximum amount of time or money that should be spent on reintegration. The implementations of the polices are not very sophisticated, they simply stop the reintegration process whenever the maximum has been reached — there is, for instance, no prioritizing of objects to fetch.

The policies described in [MES95], [HKZ02] are examples of a "what" policy.

**Network/Unplugged** [Mob01] describes an "intelligent delta selection process" which is employed in a commercial synchronization tool called Network/Unplugged. The tool utilizes deltas for object synchronization, and different types of deltas can be used depending on the user needs. The available methods are block level differencing, byte level differencing and write monitor differencing. Detailed descriptions of these methods are not publically available, but from [Mob01] it appears that byte level differencing essentially works as Unix `diff` and block level differencing as `diff`, but where each token corresponds to a block of bytes. Write monitoring works by gathering a modification log, which is passed to the peer.

Each of these methods present a different level of tradeoff between bandwidth and computing resources. Block level differencing saves least bandwidth, but requires the least amount of CPU cycles. For certain files (such as databases), write monitoring saves most bandwidth, but least CPU cycles. This is thus an example of a "how" policy.

**Original Coda, IntelliMirror** The original implementation of Coda [SK92], which did not have support for weakly connected mode, exhibits a very simple policy of when to synchronize: synchronize when the device is reconnected to the network. The same type of simple policy is employed in Microsoft's IntelliMirror [Cor99] for network folders that have been made available for off-line use.

### 4.7.2 Synchronizing Similar Data

The idea of using filters (also known as "file converters") to change the format of an object is well known. Considering that filtering can be initiated automatically, e.g. running the filter whenever the source object changes, we easily see how this could be applied in a synchronization framework: in addition to the update propagation stage we add a filtering stage to the process. Systematic integration of filtering facilities in a synchronization system might be an interesting starting point for intelligent synchronization. Data filtering is, however, a vast topic on its own and has not been surveyed in this document.

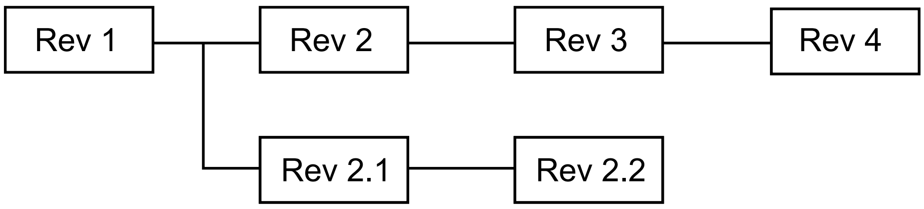


Figure 4.8: A revision tree with two branches

An interesting[*] technique for synchronizing similar data without explicit filtering is that of 3-way merging. The technique is well known among software developers, and works as follows:

Consider a the development of a software module in a single file. After the development of the initial version, two parallel branches of the module are created, with branch *A* containing some options not found in branch *B*. After a while we may have the revision tree depicted in figure 4.8. Assume that an important bug fix was added to branch *A* between revision 3 and 4 and that the bug also exists in branch *B*. We would like to apply the fix used in branch *A* to branch *B* a well.

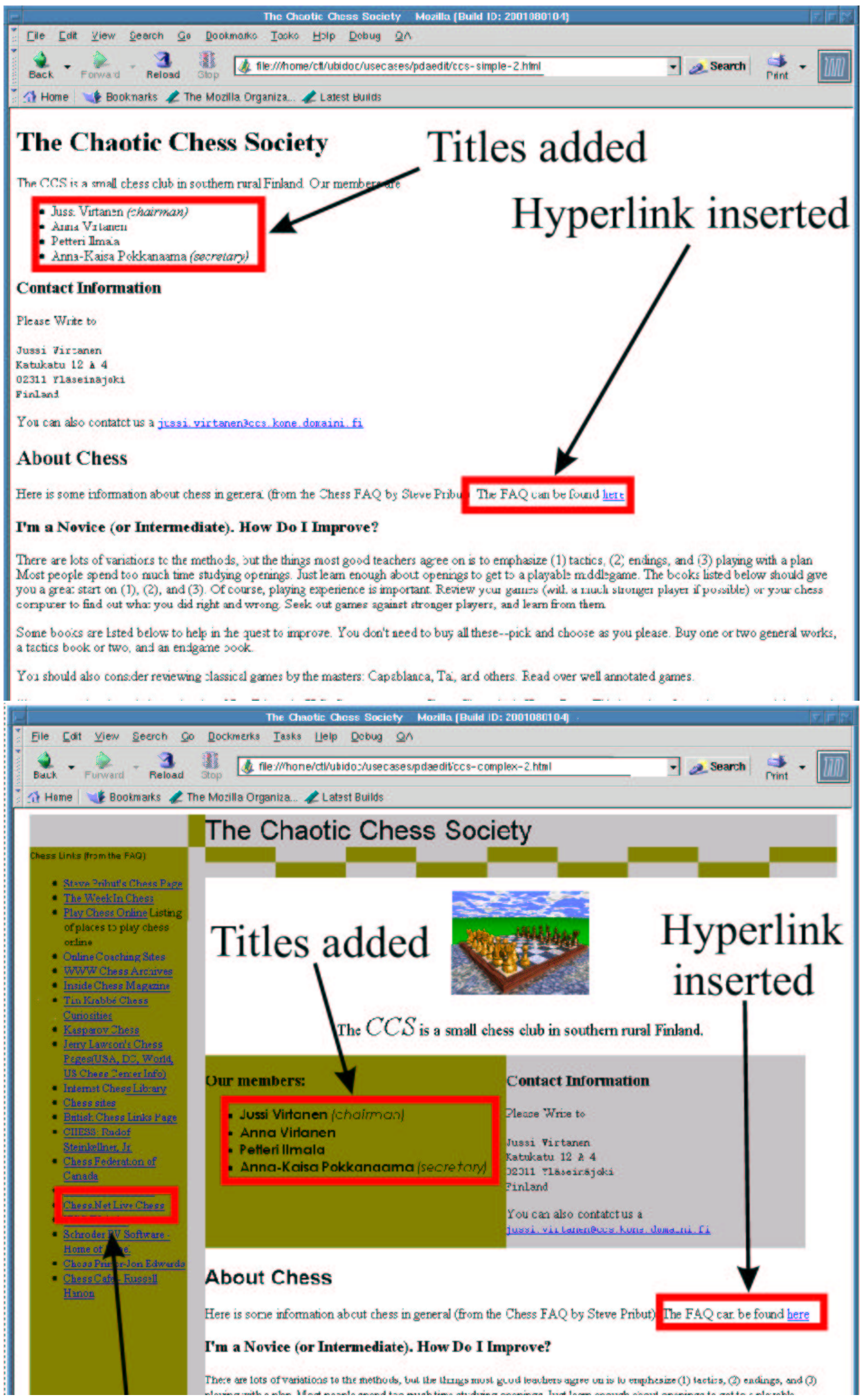
This can be done with a 3-way merge of revision 3, 4 and 2.2, which computes the changes between revisions 3 and 4, and applies them to revision 2.2. Here we also see what is meant by synchronization of similar data: revisions 3 and 2.2 are not identical, but contain sufficiently similar data for us to be able to use the bug fix from revision 4 directly on revision 2.2.

In [Lin01, 3DW] the 3dm tool capable of 3-way merging of hierarchical data in the form of unordered trees (in this case XML files) is presented. The similarity relationships between hierarchical datasets are generally more complex than those for text files, but the same idea applies: when the same substructure is found in both datasets, changes in the common substructure may be propagated from one dataset to another.

Illustratory use cases can be found in [Lin01], including an example of synchronizing two versions of a web page, one written for a constrained device and the other for desktop viewing. Changes made to the version for constrained devices are propagated to the desktop version using nothing but old and new versions of the XHTML files (no XSLT transformations, style sheets etc). Figure 4.9 illustrates this example.

Hierarchical 3-way merging and disconnected operation fit very well together. As a result of disconnected operation, there is a need to reintegrate changes, possibly from multiple concurrently modified versions. By performing successive 3-way merges one can integrate the changes from any number of modified copies of a base file. Assume the base file is *b* and

[*]In the opinion of the author, who incidentally designed the system being presented....


The Chaotic Chess Society Mozilla (Build ID: 2001080104)
File Edit View Search Go Bookmarks Tools Help Debug QA
Back Forward Reload Stop Search Print
Home Bookmarks The Mozilla Organiza... Latest Builds
Titles added
Hyperlink inserted

file:///home/ctl/ubidoc/usecases/pdaedit/ccs-simple-2.html

The Chaotic Chess Society

The CCS is a small chess club in southern rural Finland. Our members are

- Jussi Virtanen *(chairman)*
- Anna Virtanen
- Petteri Ilmala
- Anna-Kaisa Pokkanaama *(secretary)*

Contact Information

Please Write to

```
Jussi Virtanen
Katukatu 12 A 4
02311 Yläseinäjoki
Finland
```

You can also contatct us a jussi.virtanen@ccs.kone.domaini.fi

About Chess

Here is some information about chess in general (from the Chess FAQ by Steve Pribut) The FAQ can be found here

I'm a Novice (or Intermediate). How Do I Improve?

There are lots of variations to the methods, but the things most good teachers agree on is to emphasize (1) tactics, (2) endings, and (3) playing with a plan. Most people spend too much time studying openings. Just learn enough about openings to get to a playable middlegame. The books listed below should give you a great start on (1), (2), and (3). Of course, playing experience is important. Review your games (with a much stronger player if possible) or your chess computer to find out what you did right and wrong. Seek out games against stronger players, and learn from them.

Some books are listed below to help in the quest to improve. You don't need to buy all these--pick and choose as you please. Buy one or two general works, a tactics book or two, and an endgame book.

You should also consider reviewing classical games by the masters: Capablanca, Tal, and others. Read over well annotated games.

file:///home/ctl/ubidoc/usecases/pdaedit/ccs-complex-2.html

The Chaotic Chess Society

Chess Links (from the FAQ)

- Steve Pribut's Chess Page
- The Week In Chess
- Play Chess Online Listing of places to play chess online
- Online Coaching Sites
- WWW Chess Archives
- Inside Chess Magazine
- Tim Krabbé Chess Curiosities
- Kasparov Chess
- Jerry Lawson's Chess Pages(USA, DC, World, US Chess Center Info)
- Internet Chess Library
- Chess sites
- British Chess Links Page
- CHESS: Rudolf Steinkellner, Jr.
- Chess Federation of Canada
- ChessNet Live Chess
- Schroder BV Software - Home of ...e
- Chess Pri...r-Jon Edwards
- Chess Caf... Russell Hanon

The *CCS* is a small chess club in southern rural Finland.

Our members:

- Jussi Virtanen *(chairman)*
- Anna Virtanen
- Petteri Ilmala
- Anna-Kaisa Pokkanaama *(secretary)*

Contact Information

Please Write to

```
Jussi Virtanen
Katukatu 12 A 4
02311 Yläseinäjoki
Finland
```

You can also contatct us a jussi.virtanen@ccs.kone.domaini.fi

About Chess

Here is some information about chess in general (from the Chess FAQ by Steve Pribut) The FAQ can be found here

I'm a Novice (or Intermediate). How Do I Improve?

There are lots of variations to the methods, but the things most good teachers agree on is to emphasize (1) tactics, (2) endings, and (3)

Figure 4.9: Example of 3-way merging of XHTML documents. The changes in the upper page are propagated to the lower page using 3-way merging.

the modified copies are $b_1, \dots, b_n$. A file which incorporates all modifications can be obtained as $m(b, \dots m(b, m(b, b_1, b_2), b_3) \dots, b_n)$, where $m$ denotes the 3-way merging operation.

Furthermore, in addition to the common insert/update/delete operations, 3dm supports subtree copy and move operations. These operations are very important for the successful reintegration of hierarchical data. Consider for instance that a subtree has been moved in one version $v_1$, and a node in the subtree has been updated in another version $v_2$. A reasonable way of integrating these two changes is to move the subtree as well as update the node in the subtree. However, if no move operation is available, the subtree move has to be expressed as node deletions and insertions, leading to an delete/update conflict between $v_1$ and $v_2$. Note that this is indeed due to a limitation of the available operations: if we would actually had deleted the subtree and inserted new nodes somewhere else, the change log would be identical to a subtree move, but in this case a conflict should occur.

The ability to recognize subtree move and copy operations in addition with the 3-way merging ability of 3dm allows it to reintegrate tree structures that have undergone arbitrary edits, consisting of insert, delete, update, move and copy operations in a reasonable[*] manner. If reintegration fails, the tool also is able to generate detailed information about the conflicting operations.

## 4.8 Delta Transfers

In the case of data synchronization, it is very common that a previous version of the data we want to transmit exists at the receiving end. Frequently, the previous version is also quite similar to the new version, opening up the possibility for *delta transfers*, i.e. transmitting only the changes between the previous and new version, and subsequently reconstructing the new version on the receiving end using the old version and these changes.

The systems discussed do indeed make use of delta transfers on some level: Coda only transmits modified files upon reconnection (instead of blindly copying the entire file tree), as does InterMezzo. Bayou servers send write logs instead of the entire database when synchronizing with their peers. Delta transfers can, however, potentially be used to a much higher degree in these systems, by utilizing delta transfers for individual files (binary objects) as well.

In [HKZ02] a modification to Coda is presented where file updates are handled by delta transfers. The deltas between files are generated by running the well-known GNU diff utility on the old and new versions, the old version being known to exist at the receiving end. The output of diff is

[*]"Reasonable" merging behavior is defined in detail in [Lin01]. Integrating a move and update as a move combined with an update is an example of this reasonable behavior.

transmitted over the network, where it is used to generate the new version. Experiments on text files in [HKZ02] indicate 65–95% savings in the number of bytes transferred between client and server. Although these numbers are overly optimistic (no transfers of binary files, on which diff performs much worse, were included) they show the potential of delta transfers on the file level.

A file synchronization algorithm explicitly developed for high latency, low bandwidth links is presented in [Tri99]. The algorithm, implemented in the freely available rsync tool, can be used on both binary and text files, and does not assume that a particular version of the target file exists or is retrievable on the receiving end, as is the case in [HKZ02]. However, the more similar the existing version on the receiving end is, the more bandwidth can be saved.

Assume that we have two devices $A$ and $B$, and we want to transmit a file $F$ from $A$ to $B$. On $B$ exists a file $F\prime$, which is somehow related to $F$ (e.g. $F$ is a newer version of $F'$). The basic[*] rsync algorithm consists of three stages:

1. $B$ splits $F'$ into blocks and calculates a message digest[†] for each block. Let the digests be $S_1 \ldots S_i$. The digests are transmitted to $A$.

2. On $A$, $F$ is scanned for blocks whose digest match one of the digests $S_1 \ldots S_i$. As a result, we can express $F$ as a sequence of verbatim bytes from $F$ interleaved with references to matching block digests. This sequence is transmitted to $B$.

3. $B$ now constructs $F$ using $F'$ and the received sequence. When verbatim bytes are read from the sequence, these are written directly to $F$. When a reference to a matching digest is read, the corresponding block from $F'$ is written to $F$.

Thus, if for instance both files are binary files, and $F$ would be identical to $b_1 \ldots b_k F'$, where $b_1 \ldots b_k$ are some bytes inserted at the start of $F'$, the algorithm would transfer the sequence $b_1 \ldots b_k$ followed by references to the digests $S_1 \ldots S_i$. Assuming a block size of 1024 bytes, 4 bytes to index the digests, $k = 1024$ and that the size of $F'$ is 1M, roughly 5k of data would be transmitted instead of 1M. In contrast to GNU diff, the algorithm is also capable of handling moves of data (which show up as block reorderings).

In practice rsync may perform much worse due to small but scattered changes (block matches are destroyed by a few nonmatching bytes), as well as compressed data (changes in data at position $i$ may affect all bytes emitted by a compressor after it has passed over position $i$). Some interesting thoughts on how to remedy this problem are given in [Tri99].

[*]For brevity, several important details have been omitted here, such as the use of two types of message digests.

[†]The message digest is much shorter than the block. A block may be e.g. 1 kb while the digest is 16 bytes (the size of the MD4 digest originally used).

## 4.9 Conclusions

Among the reviewed systems there is none that would readily fit the role of an mobile distributed info base. Each system has its strengths and weaknesses. To design an info base for operation in a mobile environment we need to combine the best aspects of these systems. Coda and InterMezzo appear to be the most promising starting points. In Coda's case there is a certain maturity emerging from over a decade of research, the fact that it addresses several of the concerns of the mobile environment, as well as freely available source coda. InterMezzo, being derived from Coda, should have many of its strengths, as well as a simpler design, well suited for mobile devices.

OceanStore, while being an interesting concept with many good ideas, is in practice not mature yet to be used as a starting point. Bayou is especially interesting in the sense that it is the only system working as a shared database (as opposed to a shared file system). If it appears more useful to develop a database type of info base, Bayou is the natural starting point. The source code for Bayou is, however, not publically available, requiring considerable amount of programming to even get a base system up and running.

It should noticed that there are several design features that all of the systems have in common. If designing an info base on our own, we should carefully consider deviating from these:

- Acceptance of the tradeoff between strong connectivity and consistency versus weak connectivity and consistency. There is an agreement that strong consistency on weak connections is infeasible.
- Use of aggressive caching. Indeed, it is hard to imagine providing high availability in disconnected mode without large caches.
- Support for write access while disconnected. Read-only operation during disconnection is not a viable alternative.
- The use of optimistic locking, with subsequent reintegration and conflict resolution. Write or read locking is not used in any of the systems to avoid conflicts.

Some issues in this review appear to be orthogonal, and it should thus be possible to "pick and choose" the best approaches from different systems. As an example of this, consider e.g. cache coherency approaches and object transfer mechanisms: it should be possible to modify the object transfer mechanism (by using such techniques as delta transfers) without affecting the cache coherency mechanism.

An important question, which this review touches only the surface of, is that of storing data mostly as XML. What are the features that can be built into an info base, if most of the data is XML? Are there any advantages to be gained? Especially the 3DM tool should prove interesting in this context.

Furthermore, we need to address the question of whether to base the info base design on a database or a file system. These approaches have different optimal usage scenarios. Databases are more suited for retrieving and storing single records in a vast dataset whereas file systems are more efficient and easy to use for traditional "document-based" applications, such as text editors, drawing tools etc. An interesting approach would be to combine both, e.g. by allowing the file system to be aware of the structure of files containing XML data.

At least two possible paths of continued research can be envisioned:

1. Developing the underlying synchronizing file system or database, but not yet address such issues as synchronization policies.

2. Concentrate on developing policies. A synchronization policy engine would be constructed on top of an existing prototype info base, such as e.g. a combination of Coda and SyncML.

Naturally, combinations of these are also possible. It would for instance seem to make sense to pursue the first alternative as far as to have a prototyping platform, after which one could try approaching the issue of policies.

# Bibliography

[3DW] 3dm development web site. http://tdm.berlios.de. 20

[Bar01] D. Bartlett. CORBA junction: CORBA 3.0 notification service. *IBM developerWorks*, 2001. 2.3.4, 2.3.5

[Bar02] M. Bar. Keeping in sync. *Byte Magazine*, January 2002. 4.3.3

[BaW] Bayou homepage. http://www2.parc.com/csl/projects/bayou/. 4.5

[BBHS] P. Braam, R. Baron, J. Harkes, and M. Schnieder. *The Coda HOWTO (version 1.00)*. http://www.coda.cs.cmu.edu/doc/html/coda-howto.html. 4.2

[BKS+99] G. Banavar, M. Kaplan, K. Shaw, R.E. Strom, D.C. Sturman, and Wei Tao. Information flow based event distribution middleware. In W. Sun, S.T. Chanson, D. Tygar, and P. Dasgupta, editors, *ICDCS Workshop on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, 1999. 2.3.10

[Blo70] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *In Communications of the ACM*, 13(7):422–426, July 1970. 4.4.1

[BMH+00] J. Bacon, K. Moody, R. Hayton, et al. Generic support for distributed applications. *IEEE Computer*, March 2000. (document), 2.1, 2.2.5, 2.3.7, 2.12

[BNFT00] G. Bricconi, E. Di Nitto, A. Fuggetta, and E. Tracanella. Analyzing the behavior of event dispatching systems through simulation. In *In the Proceedings of the 7th International Conference on High Performance Computing IEEE*, 2000. 2.3.10

[BNT00] G. Bricconi, E. Di Nitto, and E. Tracanella. Issues in analyzing the behavior of event dispatching systems. In *In the Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10)*, 2000. 2.3.10

[Bra98] P. Braam. The coda distributed file system. *Linux Journal*, (50), June 1998. 4.2

[Bra01] P. Braam. Removing bottlenecks in distributed filesystems: Coda and intermezzo as examples, July 2001. `http://www.inter-mezzo.org/docs/bottlenecks.pdf`. 4.3, 4.3.3

[Bra02] P. Braam. Intermezzo: File synchronization with intersync, ver 0.9.3, March 2002. `http://www.inter-mezzo.org/docs/intersync.pdf`. (document), 4.3, 4.2, 4.3.1, *, 4.3.3

[CDW01] Antonio Carzaniga, Jing Deng, and Alexander L. Wolf. Fast forwarding for content-based networking. Technical Report CU-CS-922-01, Department of Computer Science, University of Colorado, November 2001. 2.3.8

[CN01] G. Cugola and E. Nitto. Using a publish/subscribe middleware to support mobile computing, September 2001. 2.1, 2.3.10, 2.3.10

[CNF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, pages 827–850, September 2001. Vol 27, No 9. 2.3.10, 2.3.10

[CNP00] G. Cugola, E. D. Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *In Workshop su Sistemi Distribuiti: Algorithmi, Architectture e Linguaggi (WSDAAL)*, 2000. 2.3.10

[Cod] Coda web site. `http://www.coda.cs.cmu.edu`. 4.2

[Cor99] Microsoft Corporation. *Microsoft Windows 2000 Server, Introduction to IntelliMirror Management Technologies. White paper*, 1999. 4.7.1

[CRW99] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, October 1999. revised May 2000. 2.3.8, 2.3.8

[CW01] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, AZ, October 2001. 2.2.4

[D$^+$94] A. Demers et al. Session guarantees for weakly-consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, September 1994. 4.5, 15

[E$^+$97] W. K. Edwards et al. Designing and implementing asynchronous collaborative applications with bayou. In *Proceedings of 10th ACM Symposium on User Interface Software and Technology*, October 1997. (document), 4.5, 4.7, 15

[EBS01] G. Eisenhauer, F. Bustamante, and K. Schwan. A middleware toolkit for client-initiated service specialization. In *ACM SIGOPS*, volume 35, pages 7–20. College of Computing, Georgia Institute of Technology, April 2001. 2.3.10

[Ere00] J. R. Erenkrantz. Handling hierarchical events in an internet-scale event service, March 2000. `http://www.ucf.ics.uci.edu/~jerenk/siena-xml/SienaPaper.html`. 2.3.8

[GCSO01] Pradeep Gore, Ron Cytron, Douglas Schmidt, and Carlos O'Ryan. Designing and optimizing a scalable CORBA notification service. 36(8):196–204, August 2001. (document), 2.2.2, 2.9

[GS00] Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. In *Ninth International World Wide Web Conference*, May 2000. `http://www9.org/w9cdrom/154/154.html`. 3.4.4

[Hei01] Dennis Heimbigner. Adapting publish/subscribe middleware to achieve gnutella-like functionality. In *Coordination Models, Languages and Applications, Special Track at 2001 ACM Symposium on Applied Computing (SAC 2001)*, 2001. 2.3.8

[HKZ02] A. S. Helal, A. Khushraj, and J. Zhang. Incremental hoarding and reintegration in mobile environments. In *Proceedings of the 2002 Symposium on Applications and the Internet (SAINT)*, 2002. 4.7.1, 4.8

[HP94] J.S. Heidemann and G.J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994. 4.3.1

[IBM02a] IBM. *Gryphon: Publish/subscribe over public networks.*, December 2002. [whitepaper] `http://www.research.ibm.com/gryphon/Gryphon/Gryphon-Overview.pdf`. 2.2.3

[IBM02b] IBM. *MQSeries Everyplace for Multiplatforms Version 1, Release 2*, 2002. [whitepaper] `http://www-3.ibm.com/software/ts/mqseries/everyplace/v12/whitepaper.html`. 2.3.10

[IET02] IETF. *Pragmatic General Multicast (PGM) protocol*, 2002. [Internet Draft] `http://www.ietf.org/internet-drafts/draft-speakman-pgm-spec-06.txt`. 2.3.10

[IMW] Intermezzo web site. http://www.inter-mezzo.org. 4.3

[K$^+$00] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM Asplos*, 2000. (document), 4.4, ‡, §, 4.3, 4.4.1, 4.4.2, 4.4.3

[Kis01] Roman Kiss. *Using the COM+ Event System in .Net Applications*, 2001. http://www.codeproject.com/useritems/solutionlcenotification.asp. 2.3.10

[Lin01] T. Lindholm. A 3-way merging algorithm for synchronizing ordered trees — the 3dm merging and differencing tool for xml. Master's thesis, Helsinki University of Technology, Dept. of Computer Science, September 2001. http://www.cs.hut.fi/~ctl/3dm/thesis.pdf. 20, *

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, 1982. 12

[Mei00] R. Meier. State of the art review of distributed event models. *IEEE Computer*, 2000. http://citeseer.nj.nec.com/437791.html. 2.2.4, 2.3.1

[MES95] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995. 4.2, 4.7.1

[Mic99] Microsoft. Message queuing on Windows CE. *Microsoft Systems Journal*, June 1999. Windows CE Developers Conference, http://www.microsoft.com/msmq/downloads/devcon99.ppt. 2.3.10, 2.3.10

[Mic02] Microsoft. *Message Queuing in Windows XP: New Features*, 2002. [whitepaper] http://www.microsoft.com/msmq/MSMQ3.0_whitepaper_draft.doc. 2.3.10, 2.3.10, 2.3.10, 2.3.10

[Mob01] Mobiliti inc. *Overview of Intelligent Delta Selection Process (iDESP)*, August 2001. http://www.mobiliti.com/PDF/iDESPOverview30.pdf. 4.7.1

[MS91] H. Mashburn and M. Satyanarayanan. *RVM: Recoverable Virtual Memory User Manual*, April 1991. 4.2.1

[OMG01a] Object Management Group. *CORBA Event Service Specification v.1.1.*, March 2001. 2.3.4, 2.3.4

[OMG01b] Object Management Group. *CORBA Notification Service Specification v.1.0.*, March 2001. 2.3.5

[OMG02a] Object Management Group. *Joint Initial Submission regarding the JMS Notification Service RFP*, 2002. telecom/02-01-02. 2.3.3

[OMG02b] Object Management Group. *Management of Event Domains Specification*, August 2002. Final Adopted Specification. 2.3.6

[OR02] Eamon O'Tuathail and Marshall T. Rose. *Using SOAP in BEEP*, January 2002. [Internet-Draft] `http://www.ietf.org/internet-drafts/draft-etal-beep-soap-06.txt`. 3.4.3

[OSW] Oceanstore web site. `http://oceanstore.cs.berkeley.edu`. 4.4, 4.4.4

[P$^+$83] D.S. Parker et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3), May 1983. †

[Pab02] C. Pabla. *SyncML intensive — A beginner's look at the SyncML protocol and procedures*. IBM developerWorks, April 2002. `http://www-106.ibm.com/developerworks/wireless/library/i-syncml2/`. 4.6

[Pla99] David Platt. The COM+ event service eases the pain of publishing and subscribing to data. *Microsoft Systems Journal*, September 1999. `http://www.microsoft.com/msj/defaultframe.asp?page=/msj/0999/com+event/com+event.htm`. 2.3.10, 2.3.10

[Pri01] Prism Technologies. *Notification Service whitepaper*, May 2001. `http://www.prismtechnologies.com/English/Products/CORBA/whitepapers/html/1notification/Notification_final_may_01.html`. (document), 2.3.3, 2.4

[PRR97] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA*, June 1997. 4.4.1

[PST$^+$97] K. Peterson, M. Spreitzer, D. Terry, M. Theimer, and A. J. Demers. Flexible update propagation in weakly consistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 288–301, September 1997. (document), 4.5, 4.6, 4.5.2

[R$^+$01a] Bill Ray et al. *Professional Java Mobile Programming*. Wrox Press, 2001. 2.3.3

[R$^+$01b] S. Rhea et al. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September 2001. 4.4, ‡, 4.4.1, 4.4.3, 4.4.4

[Ros01a] Marshall T. Rose. *RFC 3080: The Blocks Extensible Exchange Protocol Core*, March 2001. http://www.ietf.org/rfc/rfc3080.txt. 3.4.3

[Ros01b] D. Rosenblum. A tour of Siena, an interoperability infrastructure for internet-scale distributed architectures. In *Ground System Architectures Workshop (GSAW2001)*, February 2001. http://sunset.usc.edu/GSAW/gsaw2001/SESSION3/Siena.pdf. 2.3.8

[S$^+$90] M. Satyanaraynan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990. 4.2

[S$^+$97] M. J. Spreitzer et al. Dealing with server corruption in weakly consistent, replicated data systems. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, pages 234–240, September 1997. 4.5.2

[SAS01] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness — transparent information delivery for mobile and invisible computing. In *CCGrid 2001 IEEE International Symposium on Cluster Computing and the Grid, 15-18 May 2001, Brisbane, Australia*. IEEE, May 2001. 2.1, 2.3.9

[Sat96] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1):26–33, 1996. 4.2

[Sie99] J. Siegel. An overview of CORBA 3. In *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, July 1999. 2.3.4

[SK92] M. Satyanarayanan and J. Kistler. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992. 4.2, 4.7.1

[Sri01] Paddy Srinivas. *Introduction to COM+ events*, 2001. http://www.idevresource.com/com/library/articles/com+eventsintro.asp. 2.3.10

[Ste01] S. Stemberger. *Syncing data — an introduction to SyncML*. IBM developerWorks, October 2001. http://www-106.ibm.com/developerworks/wireless/library/wi-syncml/. 4.6.1

[Sun01] Sun Microsystems. *Java Message Service Documentation*, June 2001. telecom/02-01-02. 2.1, 2.3.3

[Syn00] SyncML Initiative ltd. *Building an industry-wide mobile data synchronization protocol: SyncML white paper*, June 2000. http://www.syncml.org/download/whitepaper.pdf. 4.1

[Syn02a] SyncML Initiative ltd. *SyncML Representation Protocol, v1.1*, February 2002. http://www.syncml.org/docs/syncml_represent_v11_20020215.pdf. 4.6

[Syn02b] SyncML Initiative ltd. *SyncML Sync Protocol, v1.1*, February 2002. http://www.syncml.org/docs/syncml_sync_protocol_v11_20020215.pdf. 4.6, 4.6.1

[SyW] Syncml web site. http://www.syncml.org. 4.6

[Tri99] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999. 4.8, 22

[W3C99] World Wide Web Consortium (W3C). *WAP Binary XML Content Format*, June 1999. [Note] http://www.w3.org/TR/wbxml/. 3.4.4

[W3C00a] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0*, 2nd edition, October 2000. [Recommendation] http://www.w3.org/TR/2000/REC-xml-20001006/. 3.1

[W3C00b] World Wide Web Consortium (W3C). *Simple Object Access Protocol (SOAP) 1.1*, May 2000. [Note] http://www.w3.org/TR/SOAP/. 3.3.1

[W3C01a] World Wide Web Consortium (W3C). *XML Protocol Usage Scenarios*, December 2001. [Working Draft] http://www.w3.org/TR/2001/WD-xmlp-scenarios-20011217/. 3.3.2

[W3C01b] World Wide Web Consortium (W3C). *XML Schema Part 1: Structures*, May 2001. [Recommendation] http://www.w3.org/TR/xmlschema-1/. 3.1

[W3C01c] World Wide Web Consortium (W3C). *XML Schema Part 2: Datatypes*, May 2001. [Recommendation] http://www.w3.org/TR/xmlschema-2/. 3.1

[W3C01d] World Wide Web Consortium (W3C). *SOAP Security Extensions: Digital Signature*, February 2001. [Note] http://www.w3.org/TR/SOAP-dsig/. 3.3.2

[W3C01e] World Wide Web Consortium (W3C). *SOAP Version 1.2 Part 1: Messaging Framework*, December 2001. [Working Draft] http://www.w3.org/TR/2001/WD-soap12-part1-20011217/. 3.3

[W3C01f] World Wide Web Consortium (W3C). *SOAP Version 1.2 Part 2: Adjuncts*, December 2001. [Working Draft] http://www.w3.org/TR/2001/WD-soap12-part2-20011217/. 3.3

[W3C01g] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL) 1.1*, March 2001. [Note] http://www.w3.org/TR/wsdl. 3.2

[Win99] Dave Winer. *XML-RPC Specification*, October 1999. http://www.xmlrpc.com/spec. 3.3.1

[Woo] D. Woodhouse, Red Hat Inc. Jffs: The journaling flash file system. Presented at the Ottawa Linux Symposium 2001. http://sources.redhat.com/jffs2/jffs2.pdf. 4.3.2

[ZKJ01] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001. 4.4.1