The Federated Event Service

Conor Ryan

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

September 2003

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Conor Ryan

15th September 2003

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Conor Ryan

15th September 2003

Acknowledgements

To my supervisor Prof. Vinny Cahill and to René Meier, for ideas, help, advice, reviews, patience and guidance throughout this project, thank you.

To my wife Helen, for her unending love, encouragement and support, thank you. Special thanks for reviewing my work.

I would like to thank my parents and parents-in-law for their love and support, during this demanding year.

Finally, to the M.Sc. NDS class of 2003 – the nicest, friendliest bunch of people I've ever had to work with, thank you. Please don't ever change.

Abstract

Event services provide asynchronous, decoupled, anonymous message-based communication. This facilitates scalable distributed systems composed of autonomous concurrently executing entities. There are several kinds of event services in existence addressing wide ranging issues such as Internet scale, quality of service, mobility and location awareness. When integrating systems that use disparate event services it may be necessary to inter-work their event services to facilitate communication between the systems. A federated service is a collection of autonomous concurrent services that may be linked together to provide a single logical service.

There is currently no standard solution available for heterogeneous event service inter-working. In the absence of a standard solution, system developers are forced to roll their own solutions. This is problematic as such solutions can cost time, money and effort. These solutions may be sub-optimal since developers, unless they are experts in event systems and event system inter-working, may not have considered or understood all of the issues involved.

Primarily, this project examines the thesis that a standard mechanism for federating heterogeneous event services is a valuable solution for addressing this event-service inter-working problem. A secondary aim of the project is to investigate whether such a mechanism is a viable alternative to bespoke solutions for building or extending event-based systems, when requirements cannot be met by a single event service.

To this end, the design, implementation and evaluation of such a mechanism, called the Federated Event Service (FES), was carried out. A test application that federates three different kinds of event services was also built. It was determined that the FES approach is an adequate and cost effective solution for many inter-working requirements, but federation in general cannot address end-to-end event context integrity.

Contents

List of Abbreviations

CES	CORBA Event Service
CNS	CORBA Notification Service
JMS	Java Message Service
FES	Federated Event Service
COSMIC	A real time event model for the CAN-Bus
QoS	Quality of service
API	Application Program Interface

List of Figures

List of Tables

Chapter 1. Introduction

Event services provide asynchronous, decoupled, anonymous message-based communication. This facilitates flexible, scalable distributed systems composed of autonomous concurrently executing entities. Entities are decoupled and are not required to operate in lockstep.

There are several kinds of event services in existence, addressing wide-ranging issues including Internet scale [6], quality of service [8], mobility and location awareness [5]. Event services are used in distributed systems development today in a number of areas including [33]:

- Telecommunications network management systems. Event services are used to route device alarm information and other information to network management applications and other interested users.
- Financial stock-price information systems. Event services are used to propagate stock quotes to financial applications.

1.1 The event service inter-working problem

When integrating systems that use event services it may be necessary to inter-work their event services. For example, it may be required to inter-work one system's Java Message Service [9] based system with another system's CORBA Notification Service [8] based system so that events can flow between the two systems.

There is currently no standard solution available for heterogeneous event service inter-working. In the absence of a standard solution, system developers are forced to roll their own solutions. This is problematic as such solutions can cost time, money and effort. These solutions may be sub-optimal since developers, unless they are experts in event systems and event system inter-working, may not have considered or understood all of the issues involved. This problem is compounded when inter-working of multiple kinds of event-services is required because each event service has different capabilities and interfaces.

1.2 Event service inter-working opportunities

If a standard mechanism were available for composing systems out of different kinds of event services, then system developers would have a valuable tool at their disposal for building and extending event-based systems.

For example, consider the requirement to extend the reach of a news propagation system based on an event system to mobile clients. For such a requirement, it may be viable and cheaper to inter-work the existing event service with a mobile event service, rather than by extending the existing system with a bespoke mobility solution. The standard event service composition approach might meet all or most of the requirement. At a minimum it presents developers with an possible alternative to bespoke solutions.

Novel systems may also be facilitated by such a mechanism. For example, mobile proximity aware event services, such as STEAM [5] could be combined with real time in-vehicle event services, such as COSMIC [3]. This would aid the creation of a new kind of traffic safety system where critical information such as current position, speed and braking status could be disseminated among vehicles within proximity of each other. Such information could also be conveyed over a large area by adding a scalable event service such as SIENA [6] to the mix.

1.3 Event service federation

This dissertation defines a federated service as a collection of autonomous concurrent services that may be linked together to provide a single logical service. Federating services can improve the scalability, load-balancing, fault tolerance and performance capabilities of a system. This definition has been derived from information found in [14, pp813].

Event services, due to their asynchronous, anonymous communication paradigm, are very amenable to federation. Anonymity allows the services in the federation to remain ignorant of the fact that they are federated. Asynchronous communication allows each service to operate autonomously. Therefore a federation of event services may be created transparently to the event services in the federation. The event

services may remain untouched. However, very little work has been done with respect to federating heterogeneous event services. Federation may provide a viable means of inter-working different kinds of event services.

1.4 Thesis

Primarily, this project examines the thesis that a standard mechanism for federating heterogeneous event services is a valuable solution to the event service inter-working problem.

A secondary aim of the project is to investigate whether such a mechanism can provide system developers with a viable alternative to bespoke solutions for building or extending event-based systems, when requirements cannot be met by a single event service. Is it possible to address certain event system requirements by federating different kinds of event services?

To this end, the design and implementation of such a mechanism, called the Federated Event Service, was carried out. A test application that federates three different kinds of event services, using the Federated Event Service was also built. An evaluation of the design and implementation was then carried out to determine the benefits of this solution. The evaluation also discusses the viability of federation for meeting certain kinds of system event requirements.

1.5 Goals

The aims of this project were:

- To understand the range, applicability, properties and capabilities of various important event models and services. Since there has been very little work done on heterogeneous event service inter-working, this knowledge will help to identify inter-working and federation requirements and issues.
- To determine a meta-model to classify these event models. This model will classify common attributes and functions. This common model will facilitate translation between different kinds of event models.
- To design a technology called the Federated Event Service that will allow the development of systems that integrate disparate event services. The design will

use the meta-model to define how an arbitrary event model may be mapped to and from the meta-model. Thus arbitrary event models may be mapped to and from each other.

- As proof of concept, produce an implementation of the Federated Event Service and a test/sample application to demonstrate its capabilities.
- Evaluation the design and implementation to determine the benefits of the solution for event service inter-working and to determine the viability of heterogeneous event service federation for meeting certain system event requirements.

1.6 Achievements

This project has achieved the following:

- Identified and documented many heterogeneous event service inter-working and federation issues and requirements.
- Designed the “Federated Event Service” - a mechanism for inter-working and federating heterogeneous event services. Its event model could be classified as a meta-model since it captures the necessary characteristics of many different event models for inter-working purposes.
- Implemented a proof of concept version of the “Federated Event Service”.
- Implemented a test application that demonstrates federation of the CORBA Notification Service, STEAM and SIENA.
- Provided an evaluation of federation as a valuable and realistic mechanism for event service inter-working and extension. It was concluded that the federation approach is an adequate and cost effective solution for many heterogeneous event service inter-working requirements. It was also concluded that federation is a viable alternative to bespoke solutions for meeting certain event system requirements.

1.7 Roadmap

The remaining chapters of the dissertation are summarized here:

Chapter 2 - Overview

This chapter provides the necessary background information for this dissertation. This information includes a discussion on event service concepts, an introduction to different kinds of event services, and a discussion on event service inter-working and federation.

Chapter 3 - State of the Art

This chapter covers the current state of the art for event service inter-working and federation. Current solutions and approaches are analysing to determine their characteristics, capabilities and limitations.

Chapter 4 - Issues

This chapter identifies and classifies many of the issues that must be addressed in the design of a federation of heterogeneous distributed event services.

Chapter 5 - Design

This chapter describes the high level design of the Federated Event Service including the design rationale based on an analysis of the issues identified in the previous chapter.

Chapter 6 - Implementation

This chapter describes an implementation of the Federated Event Service and discusses its features and its limitations.

Chapter 7 - Evaluation

This chapter provides an evaluation of the Federated Event Service. Strengths and weaknesses of the Federated Event Service approach compared to other approaches are discussed with respect to heterogeneous event service inter-working. The viability of federation for addressing system event requirements is also discussed.

Chapter 8 - Conclusion

This chapter provides a conclusion to the dissertation, including a summary of achievements and contributions. Areas for future work are also identified here.

Chapter 2. Overview

This chapter provides the necessary background information for this dissertation. This information includes a discussion on event service concepts; an overview of various different kinds of event services; and a discussion on event service inter-working and federation.

2.1 Basic event service concepts

An event system is a system where entities in the system communicate asynchronously by producing and consuming events. A *producer* is an entity that produces events; a *consumer* is an entity that consumes events. There may be many producer and consumer entities. An *event* is a message that contains the data that producers wish to distribute to consumers. A producer forwards events to an event service in the system. The event service is responsible for distributing events to interested consumers within the system. A consumer may specify to the event service the subset of all events that it is interested in receiving by specifying a *filter* in a *filtering language* to the event service. The event service will then only distribute the events that match the filter to the consumer. Event-based communication is generally anonymous since producers and consumers are unaware of each other – the event service decouples producers and consumers.

An entity may be a producer of events, a consumer of events or both a producer and a consumer of events. These are *roles* that an entity can take on at any time. The act of forwarding an event to an event service is known as *publishing* an event. The act of specifying a filter to an event service is known as *subscribing* for events. Hence the producer is often known as a *publisher* and the consumer is often known as a *subscriber* in event service terminology. An event service *client* is any software entity that uses the event service. For the purposes of this dissertation, a *system developer* is someone who develops systems that use event services and an *administrator* is someone who administrates, manages and configures event-based systems and event services.

Some event services require and/or support event *announcements/advertisements* by producers. Announcements notify the event service of event types that a producer may publish in the future. The event service may propagate this information to consumers. This facility allows event services and consumers to prepare resources, etc. for future event arrival.

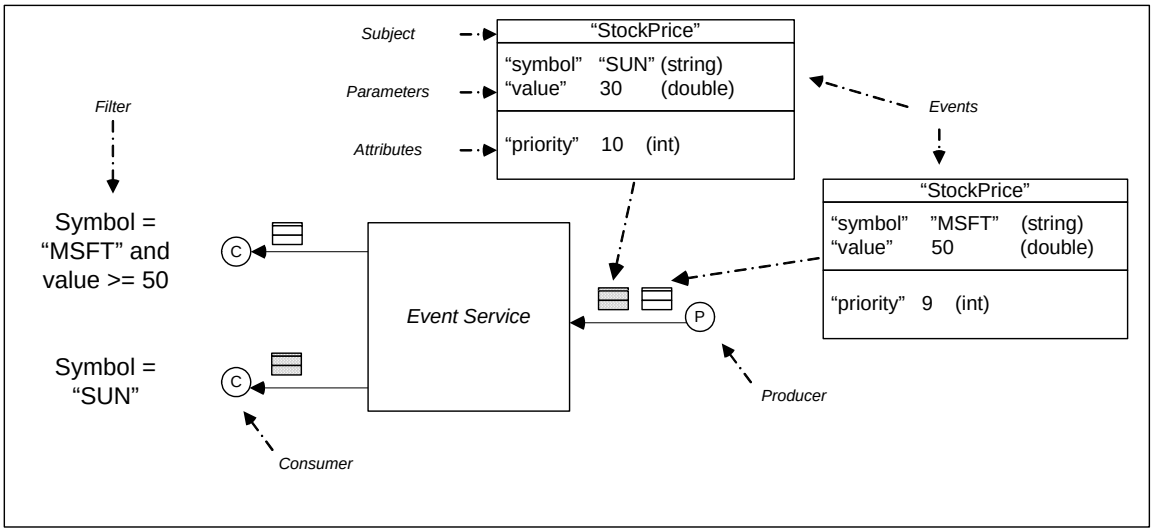

Fig 2.1 Basic Event System

Fig.2.1 shows an example of a basic event system. The system distributes stock prices to interested consumers. In this example a producer publishes two stock price events to the event service. There are two consumers, each of which has specified a different filter to the event service. One of the events is delivered to one of the consumers; the other event is delivered to the other consumer.

There are several kinds of event that may exist in an event system (see Fig. 2.2). Event services may provide support for some or all of these kinds of events.

An *un-typed* event is an event that has no well-defined structure. Such an event is very general and can contain any feasible application type and is therefore flexible to use. Only the consumer and producer of the event need to understand how to put data into the event and extract data from the event. It is opaque to the rest of the system. Such events do not easily facilitate type checking by the application. The CORBA Event Service (CES) [7] provides this kind of event.

A *typed* event is an instance of an object oriented programming language class. It has a set of attributes and a set of methods. A compiler may enforce its type statically. Many graphical user interface models use this form of event communication, for example, the Java AWT [19]. The CES is an example of a distributed event system that provides this kind of event support.

A *structured event* is a well-defined data structure into which data can be placed. Since the structure is well known any entity may examine the event to determine its contents and type of its contents. The CORBA Notification Service (CNS) and STEAM event services are examples of event services that provide this kind of event support.

> *Structured Events provide the equivalent generality and ease-of-use of un-typed event communication, while providing more strongly typed event communication.* [8, pp2-13]

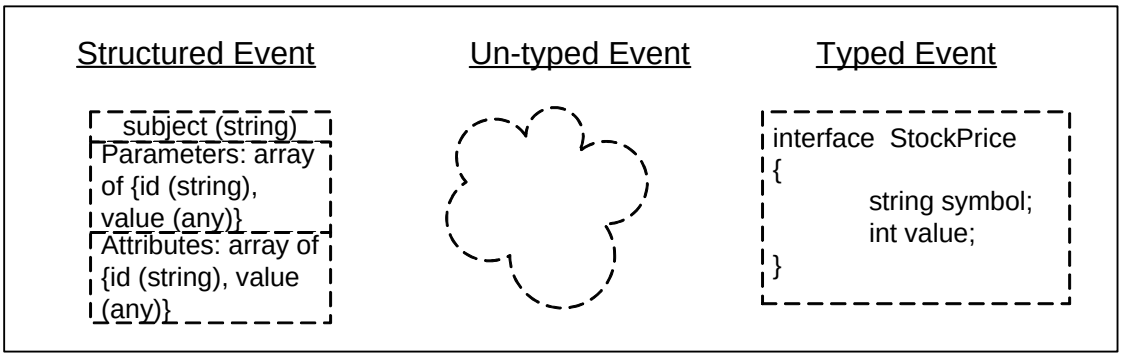

Fig 2.2 Event Types

An event contains *parameters* and *attributes*. An event parameter specifies event data. Attributes represent non-functional properties of an event such as the delivery priority of an event.

> *Attributes are related to the context, in which an event is generated like location, time, mode of operation, etc. and to quality aspects like a validity*

> *interval (expiration time) and a deadline. They represent non-functional properties of the event.* [3]

Parameters and attributes have a type and value associated with them. An event may also contain a special *subject* parameter that indicates the type of the event. Generally parameters and attributes may be accessed by identifier and/or by index.

For example, in Fig 2.1 there are two structured event instances. Both contain the subject “StockPrice” of type string; two parameters – a “symbol” parameter and a “value” parameter of types string and double respectively; and a priority attribute of type integer. The values of the parameters and attributes are different in each event instance. The event with the higher priority should be delivered to consumers before the event with the lower priority.

There are two main kinds of filtering support that may be provided by an event service. *Subject* based filtering allows events to be filtered based on an event subject value. *Content* based filtering allows events to be filtered based on the parameters values of an event.

There are two ways in which an event may be propagated from a producer to a consumer. In the *push propagation model* the producer actively forwards the event to the event service. The service then actively forwards the event to the consumer. The consumer remains passive in this situation. In the *pull propagation model* the consumer actively polls the event service for events. The event service then polls the producer for events. The propagation model may be mixed in that producers may actively push events to the event service and consumers may actively poll for those events from the event service, or the event service may actively poll for events from the producer and push these events to a passive consumer.

2.2 Taxonomy

The “Taxonomy of Distributed Event-Based Programming Systems” [1] provides a formal hierarchical classification of the fundamental properties of event based

systems. The taxonomy defines the following event system concepts that are used throughout the rest of this dissertation.

> *An event system is an application that uses an event service to carry out event-based communication.*
>
> *An event service is middleware that implements an event model, hence providing event-based communication to an event system.*
>
> *An event model consists of a set of rules describing a communication model that is based on events.*

In the next section of this chapter, examples of various kinds of event services are discussed. The discussion includes some concepts that are defined in the taxonomy. This section will give a brief description of these concepts. For the complete discussion please refer to the taxonomy.

The taxonomy defines three kinds of event models: peer-to-peer, mediator and implicit. In the *peer-to-peer* model consumers directly subscribe to producers for events. In the *mediator* model consumers subscribe to a mediator object for events and producers deliver events to the mediator object. This allows anonymous decoupled communication between producers and consumers. There may be a single mediator or multiple mediators. If the mediators are *functionally equivalent* then producers and consumers can use any of the mediators. If the mediators are *non-functionally equivalent* then producers and consumers have to deliver to or subscribe for events from the correct mediator. In the *implicit* event model consumers subscribe for specific event types rather than to another mediator or entity. Fig 2.3 shows a graphical depiction of these event model types by showing the producer-consumer dependencies.

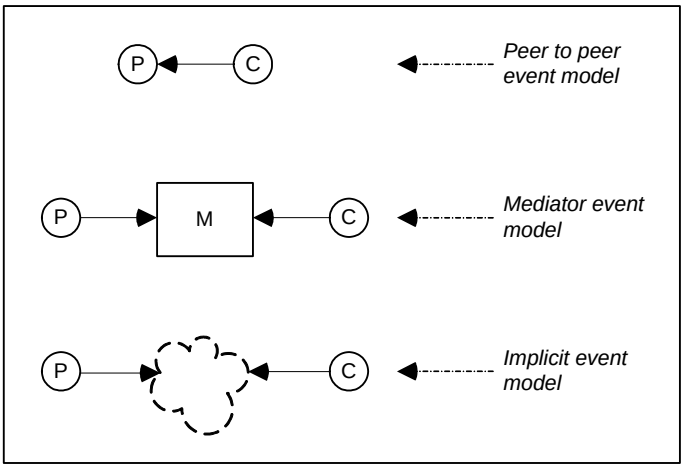

Fig 2.3 Taxonomy - Event Model Types

The taxonomy divides the event service dimension into two categories: *event service organisation* and *event service interaction*.

Event service organisation classifies an event service as either *centralised* or *distributed* based on the location of the event system entities. Distributed implies that the producers and/or consumers are located in different address spaces or physical machines to each other. Centralised implies that all entities are located in the same address space on the same physical machine. The event service middleware is *collocated* if it reside in the same address space on the same physical machine as the entities in the event system. The middleware is *separated* if it is at least partially located in a separate address spaces, possibly on separate physical machines. Separated middleware may be *single* if it is located in a single address space or *multiple* if it is located in multiple address spaces.

Event service interaction classifies the communication path over which communication between producer and consumer takes place. In the *intermediate* classification, communication must take place over at least one event service middleware component. In the *implicit* classification producers and consumers do not use an intermediate component, but use some implicit means to map events to entity addresses.

2.3 Event service survey

2.3.1 The STEAM event service

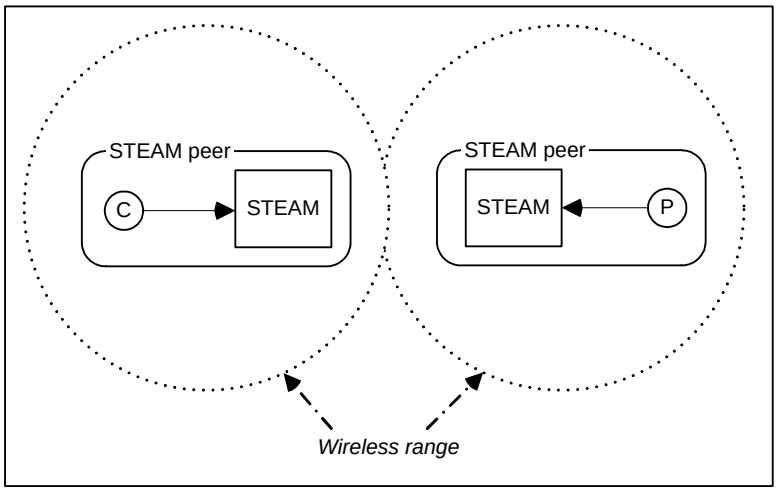

Fig 2.4 The STEAM event service

STEAM [5] is an event service that is specifically designed for mobile computing on wireless ad hoc networks. To exist in this environment the event service does not rely on any centralised service or component and is thus inherently distributed. Its design is based on that fact that entities are more likely to interact when they are in close proximity. STEAM uses the structured event type. Events are propagated with best effort reliability via the push propagation model.

STEAM employs the implicit event service interaction type. There is no intermediate component used for communication. The subject of an event is mapped to a group. Any consumer that subscribes for events with this subject (via a subject filter) implicitly joins the group. Any event produced with this event subject is forwarded to all members in the group.

In addition STEAM provides a proximity-based group communication service. To this extent, publishers specify a *proximity filter* when announcing event types that

specifies the area within which events of that type are valid. Event instances are delivered to interested consumers in the area.

Finally STEAM provides content filters that allow consumers to perform event selection based on event content. Subject and proximity filters are applied on the producer side. Content filters are applied on the consumer side only. This reduces the amount of work that producers have to perform and in conjunction with proximity filtering improves the scalability of the system.

2.3.2 The SIENA event service

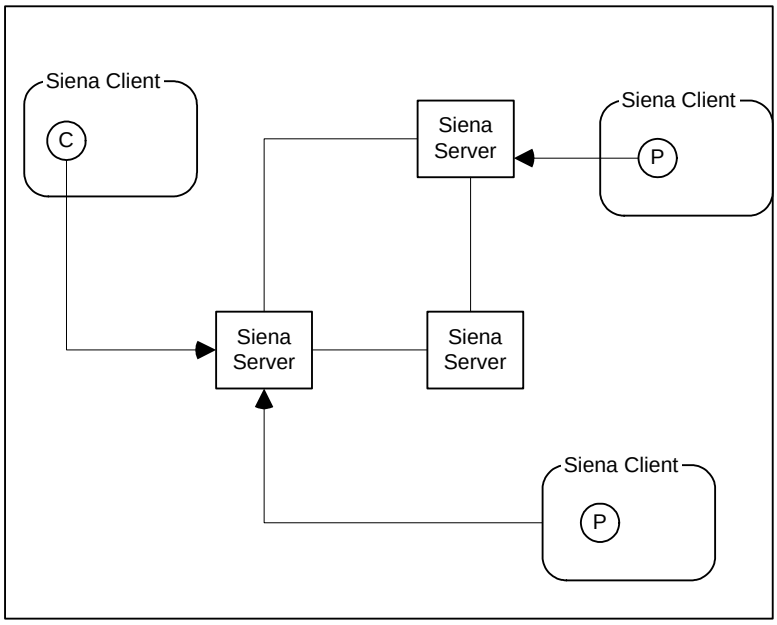

Fig 2.5 The SIENA event service

SIENA [6] is a scalable event service that is based on a distributed architecture of event services. It is specifically designed for wide area networks. SIENA provides structured events (parameters only), content filtering, and push event propagation with best effort event delivery.

SIENA implements the mediator event model. There may be one mediator or multiple functionally equivalent mediators. An event service client may use the event service through any one of its mediators.

SIENA achieves its scalability through a number of methods. The SIENA service may be distributed across a number of servers that may be organised in a hierarchical or peer-to-peer fashion. Announcements in SIENA are used to optimize the routing of subscriptions and publications. Filters are applied as physically close as possible to producers. Events are replicated as physically close as possible to consumers by means of multicast.

2.3.3 The COSMIC event service

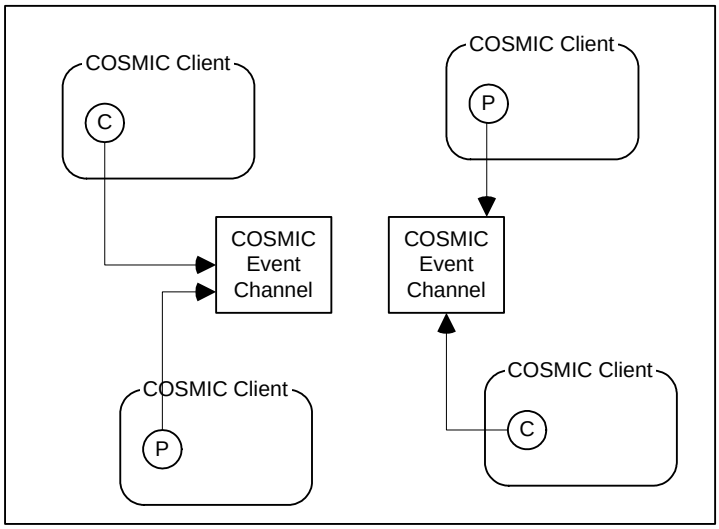

Fig 2.6 The COSMIC event service

The COSMIC [3] event service is a distributed event service that features rigorous hard real time event delivery guarantees with respect to time and reliability. It achieves these guarantees through its close coupling to a real-time CAN-Bus network. COSMIC provides un-typed events (based on the CAN 2.0B message class standard that consist of a 29 bit identifier and 8 bytes of data) and push event propagation.

COSMIC supports different event channels. An event channel disseminates all events of a certain subject. An event channel has certain non-functional attributes associated with it such as latency, dissemination constrains and reliability. There are three possible kinds of event channels. Hard real time event channels (HRTC) are considered to meet all temporal requirements under specific fault assumptions. Soft real time event channels (SRTC) are scheduled by their deadlines, but they are not

guaranteed under transient overload conditions. Non real-time event channels (NRTC) are used for events without any specified timeliness requirements in a best-effort manner. While HRTC and SRTC distribute events of restricted length to meet the responsiveness of real time systems, NRTCs may transfer bulk data in a sequence of message fragments.

Producers and consumers must be connected to the correct event channel in order to communicate. Hence, COSMIC employs the multiple, non-functionally equivalent mediator event model.

2.3.4 Other Event Services

CORBA Event Service (CES) [7]

The Common Object Request Broker Architecture (CORBA) is a specification of the Object Management Group (OMG) that defines how objects may be managed and how they interoperate under a middleware composed of Object Request Brokers (ORBs). The CORBA 2.0 specification defines a wide range of general-purpose object services, one of which is the CORBA Event Service. This service defines an event based communication mechanism for CORBA applications. The service is completely distributed; there is no dependence on global, critical, or centralised service. It uses standard IDL interfaces and does not require any extensions to CORBA. It may be implemented in many different operating environments, for example, environments that include threading and those that do not. The service is composed of event channels that broker event messages, event suppliers that supply event messages and event consumers that consume event messages.

COBRA Notification Service (CNS) [8]

The CORBA Notification Service is an extension of the CORBA Event Service. The service preserves all of the semantics of the CORBA Event service and most importantly adds filtering and quality of service facilities. The main design goal of the CNS architecture is to provide important features that are required to satisfy a variety of applications with a broad range of scalability, performance, and quality of service requirements.

Java Message Service (JMS) [9]

The Java Message Service is a specification that defines a common set of enterprise messaging services and facilities. It attempts to minimise the set of concepts a Java language programmer must learn to use enterprise messaging products. It strives to maximize the portability of messaging applications. Two means of communication are defined: point to point and publish/subscribe. An implementer of JMS need only provide one of these mechanisms. Java 2 Enterprise Edition providers must implement both mechanisms.

2.4 Event service inter-working

As explained in the introduction, system developers may be required to make different kinds of event service work together (inter-work). Inter-working requirements might include:

- Support the propagation and integrity of events between event services.
- Support the propagation of announcements and subscriptions request between event services.
- Maintain the event context of an event between event services such as event proximity, latest delivery time and priority.

Bilateral inter-working exists when two event services participate. *Multilateral* inter-working exists when three or more event services participate. Requests and events must be translated between event services in order for inter-working to occur. A one-step translation process is only really feasible for bilateral inter-working. Multilateral inter-working becomes more difficult to achieve with one step translation as the number of participants increase. With N event services, the number of translation possibilities is: $\sum_{i=1}^{N-1} i$.

Therefore two-step translation is necessary for general multilateral inter-working. With two-step translation only the ability to translate an event model to and from a common model is required. To translate between two different kinds of event models, requests and events are first translated from a source event model to the common model. They then may be translated to the other event model. With N event services, the number of translation possibilities is N.

A desirable requirement for event service inter-working is that *end-to-end* event context is maintained. This implies that the context of an event as defined by the producer of that event is maintained by event services and between event services until the event is delivered to the consumer(s) of that event. For example if a publisher specifies that an event must be delivered within the next 5 seconds then the participating event services and any inter-working infrastructure must ensure that that event is received within 5 seconds by all interested consumer(s). This context cannot be met if an event service does not provide such guarantees.

2.5 Event Service federation

This sub-section on federation and has been adapted from [14, pp813] to cover event service federation. A federated event service provides a single logical event service to clients but consists of a number of autonomous event services possibly located in different remote locations. Federated services offer a number of advantages:

- Each service in the federation provides a subset of the complete service. This arrangement improves reliability because if a single service fails, the rest of the services are still available.

- Services in a federation share the processing load of the logical service. This can improve performance and scalability because different services in the federation can work in parallel handling different sets of producers and consumers.

- Federation of a service permits you to maintain distinct administrative domains while still providing a single logical service. This facilitates easier management of a large service.

An important aspect of federation is that the services in the federation are either ignorant of the fact that they are federated, or, alternatively, each service has knowledge only of its immediate neighbouring servers and not of the federation as a

whole. This allows event services in the federation to continue to operate when other event services in the federation fail.

2.6 Inter-working possibilities

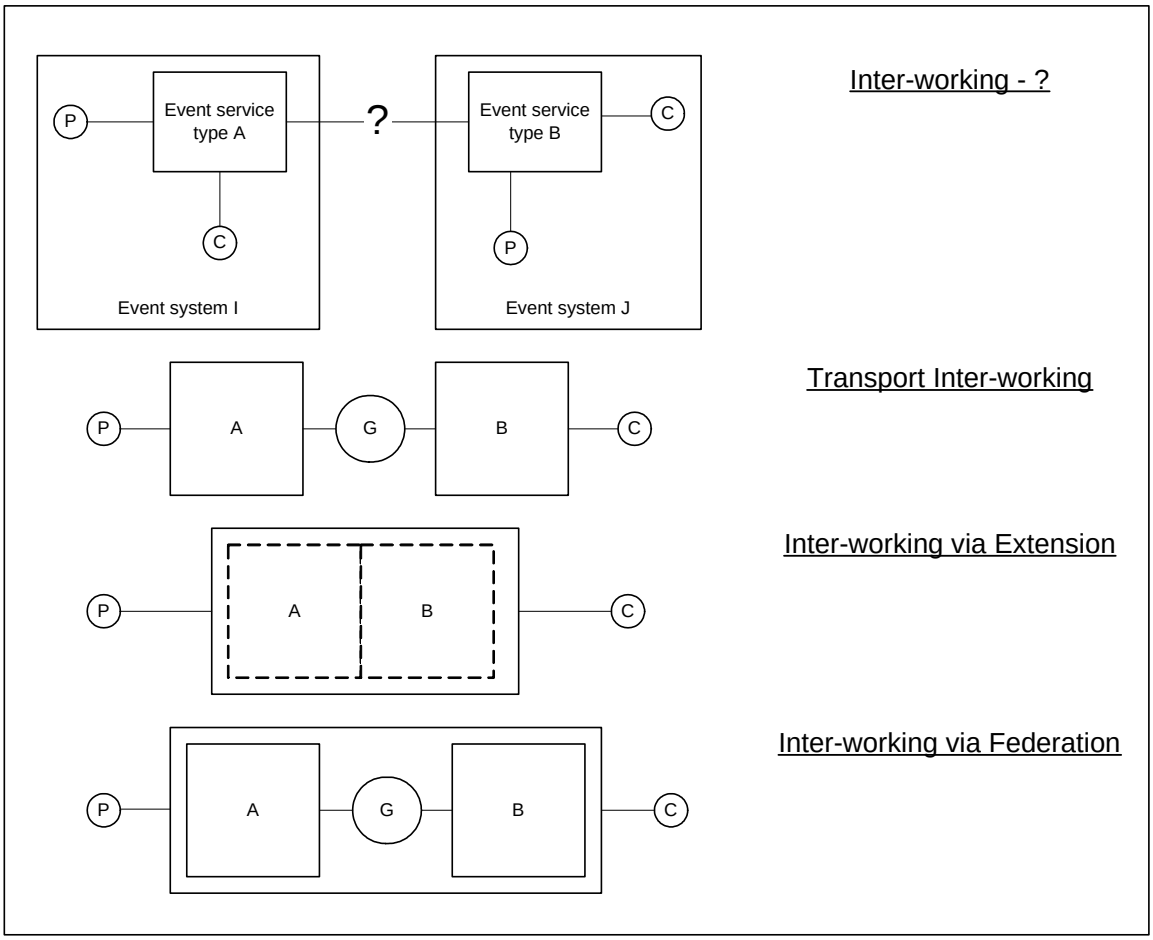

Fig 2.7 Event Service Inter-working

Fig 2.7 outlines several ways to inter-work two heterogeneous event services.

2.6.1 Transport inter-working

This approach places a gateway between the two event services. This gateway contains a producer/consumer entity. This entity subscribes to both event services for certain event types or perhaps all event types. When the entity receives an event from an event service it maps the event to the other event service's event model and publishes the event to that event service. Essentially this method provides event

mapping and transport functionality only. For example, subscription requests are not propagated between event services, so events may needlessly travel between event services. Event context information such as event priority and deadlines may be maintained between event services if there is a close match between both event models (assuming the gateway maintains the context also). This approach does not require modification of event services.

2.6.2 Inter-working via extension

Inter-working via extension extends one event service or both event services with the facilities or a sub-set of the facilities of the other event service. What results is essentially a new event service that can support both kinds of event service clients. The full functionality of both event services may be maintained.

2.6.3 Inter-working via federation

This kind of inter-working presents a single logical event service to event service clients. Like transport inter-working it requires a gateway between event services that contains a producer/consumer entity for mapping and propagating events between event services. In addition additional intelligence is required in order to propagate requests from one event service to the other event service. For example, a subscription request made at event service A for all events should be propagated to event service B. The federation should remain transparent so that existing event services and event system do not require modification and so that the federation continues to operate when any of the services in the federation fail. A general event service federation may be multilateral with event services linked in an elaborate topology. The next section outlines the requirements for such a federation.

2.7 Heterogeneous event service federation requirements

Existing examples of event service federation, as described in the state of the art chapter, do not provide a single logical view of a set of event services. In addition there are no current examples of heterogeneous event service federation. This section attempts to define a minimal set of requirements that a federation of heterogeneous event services should meet, so as to be considered a true federation. The design of the “Federated Event Service” will address these requirements.

1) The service must appear as a single logical service to the producers and consumers in the system. Therefore clients of existing systems can continue to work with little or no modifications. The service is transparent to them. They are still essentially using the same single event service model. Yet they can still use the services of any event service in the federation.

2) The service supports the propagation of event data (type data, parameter data, attribute data and subject data) between and across heterogeneous event services.

    a. Where possible the service should maintain the integrity of that data. This will not be possible to achieve in many situations:

        i. It may not be possible for the service to map between different kinds of event types to the satisfaction of the application - e.g. mapping from un-typed to structured events or mapping from one kind of structured event to another kind of structured event.

        ii. Event services may not be able to facilitate the event data types and sizes of another event service.

    For situations like this it should be possible for the administrator or system developer to specify how specific events may be mapped between specific event services.

    b. The service must maintain event data integrity when propagating events across an intermediate event service to another event service. This is necessary so that the event may be applied consistently at different event services. It may be up to the system developer or administrator to ensure that the intermediate event service(s) in their system can cope with the event size.

3) The service supports the propagation of announcement and un-announcement requests between and across heterogeneous event services. The service must also fabricate announcements if necessary for event services that require them before event publication.

4) The service supports the propagation of subscription and un-subscription requests between heterogeneous event services. This requirement is important as it prevent events from needlessly traversing event systems and consuming resources. It also improves the scalability of the system. An adequate mechanism for handling the different kinds of filtering languages supported by event models, and for mapping between these languages is required. Filters should be maintained across event services so that they may be applied consistently at each event service.

5) Where possible, the service maintains the event context between and across heterogeneous event services. For example if event services support similar capabilities then it may be possible to maintain that context. Event context might include such diverse things as event priority, event proximity, event delivery reliability and event expiration time. The data describing this context is stored in the event attributes. Attribute data is maintained as described in requirement 2 above.

Chapter 3. State of the Art

This chapter discusses currently known technologies in the area of event service inter-working and event service federation analysing characteristics, capabilities and limitations.

3.1 Notification/JMS Interworking RFP (OMG)

Due to the popularity of CNS and JMS and the need to inter-work them, the Object Management Group (OMG) has recognised the need to standardise CNS and JMS inter-working. OMG is currently working on standardizing this and has issued a request for proposal [16]. Vendors such as PrismTech and IONA have submitted proposals for this [17], [18].

The mandatory points of this RFP include:

- The ability to maintain message content integrity across the JMS and CNS.
- The ability to maintain critical QoS across the JMS and CNS.
- The ability to support (optional) transactional semantics across the JMS and CNS.

Optional points of the RFP include:

- The ability to choose between “simple” and “complex” filtering, e.g.. JMS SQL92 or CNS ETCL.

Both IONA’s and PrismTech’s first proposals to OMG use the concept of a bridge to connect a CNS event channel with a JMS topic. Message conversion is statically defined. For filtering is has been proposed to extend the CNS to support JMS’s SQL92 filtering language. It has also been proposed to extend the CNS with other features of the JMS to allow it to support JMS point-to-point functionality.

3.2 CNS/JMS Bridge (University of Mannheim)

The CNS – JMS Bridge [11] is a Java based, flexible and easily configurable bridge that enables event passing event systems based on JMS and CNS. The bridge registers

as a producer and a consumer to both event services. The advantages of this approach include:

- There is no need to modify or extend the event services.
- Existing event systems do not have to be recompiled.
- There is no need to convert filtering rules.

For event conversion to take place, it is necessary to identify the event types that correspond to each other in both systems and to develop conversion routines that perform the mapping of the respective types. Conversion in both directions may not be necessary if all producers are connected to one event service and all consumers are connected to the other event service and vice-versa.

The bridge covers mapping of structured event type information (the domain_name, type_name, event_name in CNS and the type parameter in JMS); event priority and persistence QoS characteristics; and conversion of event headers and event bodies. The bridge does not cover conversion of CNS Any events (un-typed) or CNS Typed Events.

The bridge's implementation consists of a Converter class hierarchy for conversion of event headers and bodies. A single class can generally manage event header conversion. Multiple converters might be required for each body type. Classes are further subdivided based on the direction of the conversion. Further classes determine whether a particular Converter can convert a particular event or not. The bridge derives its flexibility from this hierarchy and it can be extended with Java classes to provide additional conversion support. The class to use is chosen at run time based on the event type information. Java also gives the bridge good platform independence and portability.

3.3 OpenFusion Notification Service Connectivity Bridges (PrismTech)

These notification service connectivity bridges [20] can be used to create message bridges between the OpenFusion Notification Service and IBM's MQSeries [21] and TIBCO Rendezvous [22] message oriented middleware products. Bridges may be

created and administered either through a GUI or programmatically. Bridges may be unidirectional or bi-directional. A default message mapping is provided but developers may specify their own mapping by supplying appropriate Java code. Message delivery QoS is supported.

The architecture consists of bridge factory objects that are responsible for creating bridge objects. There is a bridge factory object for creating MQSeries bridges and a bridge factory object for creating TIBCO bridges. Mapping is performed via a mapping plug-in object that implements a specific mapping interface. Default implementations are provided and the developer can supply their own implementations Messages are tagged as they pass through a bridge. This is used to prevent messages from endlessly cycling over bi-directional bridges.

3.4 CES/CNS Federation (OMG)

The design of the CES and CNS allow event channels to be easily plugged together into a limited kind of event service federation. It does not address heterogeneous event service federation. It is up to the application to plug channels together and to assigned producers and consumers to the correct channels. It provides event transport inter-working only. Its main use is to facilitate load-balancing and performance tuning in event systems.

3.5 Analysis of the state of the art technology

Based on the state of the art study there seems to be little research into the area of federating or inter-working heterogeneous event services. CNS/CES federation does not address heterogeneity of event services. Most of the interest, for commercial reasons, lies in the inter-working of CNS and JMS.

CNS and JMS inter-working is a less difficult problem than generic event service inter-working. Firstly this is a bilateral integration issue where the inter-working of two kinds of event services need only be considered. Secondly CNS and JMS support similar event models, feature sets and event structure. Finally JMS is an abstract interface only. It does not specify implementation details. It has been designed to encapsulate the heterogeneity of event services.

The OMG Notification/JMS inter-working RFP has provided allowances to extend the CNS to support JMS requirements. The current proposals use static event mapping and conversion. This may require modifications and rebuilding of client application code, so that application events conform to the necessary structure. This solution provides event transport inter-working only. Extensions are also required to the JMS and CNS implementation to support the inter-working spec.

The University of Mannheim CNS/JMS Bridge provides very flexible event mapping and conversion support and does not require any modifications to event services or systems. Users of this bridge may need to write Java code to support their application event types. This solution provides event transport inter-working only. This bridge does not address filtering at all. Events may unnecessarily traverse the event systems consuming resources. The PrismTech OpenFusion Notification Service Connectivity Bridges is a similar solution to the University of Mannheim CNS/JMS Bridge with similar issues.

Chapter 4. Issues

This chapter identifies and structures many of the issues that must be considered in the design of a system for the federation of heterogeneous event services. Some of these issues face any system inter-working effort; others are particular to event service inter-working.

4.1 Event model heterogeneity issues

Event models differ in the many ways. For a federated service to be valuable it must cater for a wide variety of event models. The design of the service must take these differences into account when prioritising features for inclusion. Table 4.1 summarizes the event services that were studied for this dissertation. Event model, event service organisation, and event service interaction model are summarised here. These concepts are discussed in the overview and in [1]. Important event service features including propagation model support, event type support, filter support, and event service specific features are summarised here. This is not an exhaustive feature set - other features that event services may support include transaction support, event batching and atomic delivery of events.

How can a federation design handle all of these different kinds of features? Variations in propagation models may be worth supporting since there are few permutations possible. Filtering poses a difficult problem. Endless filtering languages are possible. In addition, other kinds of filtering such as proximity filtering are possible. Mobile event services introduce issues such as frequent-disconnection and event services that may connect to any point in the federation. Event services provide different QoS features such as event deadlines, order, priority and reliability guarantees. Event services may use different ranges and units to represent QoS features. These features cannot be maintained by event services that do not provide these features.

	STEAM	SIENA	CES	CNS	JMS	COSMIC
Event Model	Implicit	Single or multiple functionally equivalent mediators.	Single or multiple non-functionally equivalent mediators.	As CES.	Single mediator.	Single or multiple non-functionally equivalent mediators.
Service Organisation	Distributed. Collocated middleware.	Distributed. Separated middleware. Single or multiple.	Centralised or Distributed. Collocated or separated middleware.	As CES.	Distributed. Separated middleware.	Distributed. Separated middleware.
Service Interaction	No Intermediate Implicit addressing.	Intermediate. Centralised or distributed.	Intermediate. Centralised or distributed (via federation).	As CES.	Intermediate. Centralised.	Intermediate. Centralised.
Propagation Model Support	Push producer. Push consumer.	Push producer. Pull consumer (C++ API). Push consumer (Java API).	Push producer. Pull producer. Push consumer. Pull consumer.	As CES.	Push producer. Push consumer. Pull consumer.	Push producer. Push consumer.
Event Type Support	Structured	Structured	Un-typed Typed	Un-typed Typed Structured	Structured	Un-typed (29 bit id + 8 bytes data)
Filter Support	Subject Content Proximity	Content	None	Content (structured events)	Header properties	Subject Attribute
Other Features Supported	Mobile, ad-hoc proximity based.	Highly scalable (Internet scale).	Standard event service.	As CES. Event & event channel QoS.	Standard Java interface for event services. Transaction Support.	Hard real time event delivery guarantees.

Table 4.1 – Event service summary

4.2 Distributed system issues

For inter-working two or more event services at a single physical location, an event service federation system may not need to deal with many distributed system issues. For example using such as system to integrate a CNS system and a SIENA system could be achieved via a single process that interfaces to both the CNS and the SIENA systems. Issues such as clock synchronisation, event routing and inter-process communication do not exist. Indeed this sort of bilateral inter-working problem could be a quite common usage of an event service federation system. However in the general case an event service federation system has to deal with a myriad of network and distributed systems problems. This section elaborates on these issues.

4.2.1 Communication

The communication requirements of a federated event service can be provided by its events services. No other communication mechanism may be required. Obviously this mechanism is perfectly suitable for distributing asynchronous on-way events across processes in the system. However it may be difficult to implement the request-response requirements of a subscription request for example on this kind of communication layer. This distributed hop-by-hop communication mechanism is more subjected to failure due to individual event service failure. Allowances also have to be made for the fact that mobile event services such as STEAM may be involved in the communication path.

4.2.2 Naming

There are various mechanisms in use by event services today to identify event types and instances. The federated event service design must consider these different naming mechanisms. For example:

- The CES and CNS use event channels to distinguish event types and instances.
- SIENA and CNS allow structured events to be identified via a subscription filter.
- STEAM and COSMIC use a subject identifier to uniquely identify an event type and instance.

The design must also cope with the following naming issues:

- Ensuring that names, where necessary, are unique across the federation.
- Event services with case sensitive and case insensitive naming requirements.
- Varying maximum name lengths.

Event service federation introduces the requirements for individual event service identification and addressing. This is necessary so that subscription and announcement requests may be directed at a subset of the federation. This helps to improve the scalability of the system. Otherwise requests and responses would have to be blindly routed to all event services. Event service names must be unique to a federated event service. It is also conceivable that event services could be part of two or more federated event services. In this situation it is necessary to provide a known (possible global) unique name for the federation. Ensuring the uniqueness of event service identifiers and federation identifiers is the issue here. For federation control and administration purposes it may be necessary to identify and address individual gateways and access points to event services.

4.2.3 Time Synchronisation

To consistently apply time related QoS attributes across a distributed federation of event services a time synchronisation mechanism is required. Ad hoc wireless event service participation complicates this issue somewhat. Time synchronisation in distributed systems is discussed in [27].

4.2.4 Transparency

For inter-working existing event services using a federation system it should be possible that existing dependant event systems remain ignorant of the federation system. Event services may have multiple client applications and it may not be realistic or even possible to update them to provide support to the federation system due to cost, time or unavailability of source code. Fortunately the anonymous, decoupled communication nature of event models facilitates transparency somewhat.

4.2.5 Scalability

In addition to the event sizing issues outlined in the event model heterogeneity section event services may scale size-wise in other ways:

- Number of producers and consumers
- Number of middleware components
- Number of events that may be queued for delivery to a consumer
- Number of events that must be processed over a particular time period

Since the federation system will have to act as gateways between two or more event services it could conceivably be expected to handle the scaling requirements of these event services combined. This is a very difficult requirement to achieve without knowing in advance what scalability requirements should be provided. Therefore the system must provide mechanisms under user control to control the work that it is expected to perform. The federation system may also need to scale geographically. This may not be an important issue as it would be more prudent to address geographical scale by adding a scalable event service such as SIENA to the federation.

4.2.6 Security

When federating event system of different types, one may inadvertently expose a secure event system over an insecure event system. For example imagine that an insecure SIENA system or STEAM system was federated with a secure sensitive CNS system. An intruder could subscribe for all CNS events via the SIENA system or wirelessly via the STEAM system. Security in federated event services unveils many issues such as system wide authentication and authorisation issues. These issues will not be examined further in this thesis, and are left for future work. For a very good introduction to network security, please see [30]. For a good overview of security issues as they pertain to event services, please refer to [31].

4.2.7 Routing

Where a federated system is composed of one gateway or a set of gateways connected in a chain, no routing decisions need be made. In this situation, requests and responses

(messages) on the single input line may only ever by forward on the single output line. Routing in an event service federation raises many more issues including:

- Which routing protocol should be used? Flooding? Something better? What about routing around failed gateways?
- Can event proximity and/or a time to live value be combined with routing to decide when events and requests can be dropped?
- A diagnostics protocol may be required to allow administrators to pinpoint failed gateways and bottlenecks in the system.

4.2.8 Management/Administration/Configuration

A small federation of event services can be easily managed manually. If the federation grows to a large size then problems can arise. How can such a network be created, monitored, controlled and configured? Especially considering some of the event service clients may be mobile and widely distributed. Gateways may need substantial configuration information to be supplied to them, depending on the number of events that they need to handle and/or the kinds of mapping they need to perform and/or the number of event services they need to inter-work. Ideally configuration information should be structured in an easy to comprehend object-oriented form. Ideally a tool should be available to create and manage this information, as this is likely to be an error prone task if done manually. A standard open format such as XML should be used to record this information.

4.2.9 Fault tolerance

The remaining services in a federation should continue to operate in the presence of failure of any of its event services. However the event system may no longer be usable, if a critical event service goes down. Such failures could be masked by redundancy in software and/or hardware and the transparency of federation may facilitate such redundancy.

4.3 Other issues

4.3.1 Platform and language heterogeneity

This is a common problem in distributed systems. The federated service is expected to work with heterogeneous event services so it must cope with event services that run

on different platforms (e.g. LINUX [32] or Win32 [26]) and provides different interface languages (e.g. C++ [25], Java [15], CORBA IDL [14]).

4.3.2 Threading support

Event services may have different threading requirements. For example, they may expect all requests to be made on the same thread or they may support requests on different threads and deliver events on different threads. The federated service will have to deal with these differences.

4.3.3 Event Size

Event size limits are not often specified for event services. The system developer may not be able to determine these limits from API documentation and may have to experiment with the event service to determine its limits. The scalability section above covers more sizing issues. Things that should be specified include maximum parameter identifier length, maximum number of parameters, maximum event size, integer value range and floating-point value range. It is not possible to map an event to an event service that cannot cope with the dimensions of that event without data loss. Such an event service could not be used as an intermediate event service without the use of fragmentation in an event service federation.

4.3.4 Event Mapping

Event mapping could be user defined and/or automatic. Application developers could provide pluggable code and/or configuration information to support flexible event mapping. Different kinds of event services may have different mapping rules and may require different mapping information. Event mapping may be one to one, one to many, many to one, unidirectional an/or bi-directional.

4.3.5 Performance

The overhead of a generic federation solution including the protocol mapping and translation work may be unacceptable for certain time critical application types. Perhaps tools could be used to optimise translation work.

4.3.6 Testing

Adequately testing a federation of event services requires a large amount of test scaffolding, including test consumers and producers for the event services in the federation. Good debugging/logging facilities are required to debug and monitor the system.

4.3.7 Error Handling

How are errors handed by the system? Can the application be notified of errors in the system? What about event services that do not provide event delivery guarantee and/or acknowledgement?

4.3.8 Deployment

Accessing different kinds of event services from the same process or physical machine may lead to problems such as competition for resources such as memory, ports, processor, etc. Other problems include incompatible libraries dependencies.

4.4 Summary

This chapter throws up a lot of issues related to event service inter-working, integration and federation. This is not an exhaustive set of issues. It is interesting to note that the Internet Protocol (IP) encapsulates and addresses many of these issues to present a unified layer to the transport layer. IP deals with such issues as different kinds of networks (including fixed and wireless), routing issues, different protocols, different messages sizes, naming, different QoS and so forth. For more information on general internetworking issues and solutions please see [13, pp418].

Chapter 5. Design

The chapter documents the design of a system for federating heterogeneous event services – the "Federated Event Service" (FES). The design is centered on the FES event model. The FES is realized by a set of gateways and adapters that implement the model. The next section provides an overview of the design. The FES event mode, the FES gateways and the FES adapters are each described in their own sections. The Configuration section defines mandatory configuration information that is required for a FES system. The Use Case section illustrates the design in terms of a potential real-world use of the FES. Finally precise definitions of the gateway and adapter interfaces are given at the end of the chapter.

5.1 Overview

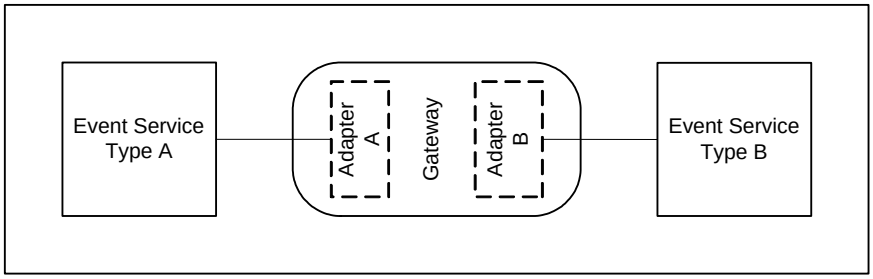

Fig 5.1 A simple FES system

A FES system consists of two or more event services and one or more *gateways* that bridge them. A gateway propagates *requests* between event services. A gateway interfaces to each event service by means of an *adapter*. An adapter maps generic FES requests to specific event service requests and vice-versa. An adapter is required for each kind of event service in a FES system. The gateways in a FES system form a completely distributed system. Each gateway is an equal peer in the system. There are no centralized points of control or failure and gateways do not maintain any global state.

A request specifies the event service where it originated from (the *source event service*), the event service(s) at which the request should be applied (*destination event service(s)*), and the request parameters. For example, a subscription request specifies

the filter that should be applied. A publication request specifies the event that should be published.

Fig 5.1 shows a static view of a simple FES system. Here event service type A is connected to event service type B by means of a gateway. If for example, the gateway receives a subscription request from event service A for event service B, it applies that subscription to event service B via adapter B. Later if the gateway receives a publication request from event service B for event service A, it then publishes that event to event service A via adapter A.

A request is encapsulated in an asynchronous *control event*. Control events are the only means by which requests may be communicated to gateways and by which gateways communicate. A gateway acts as a producer and a consumer of control events for each of the event services that it is connected to. Therefore a request may be forwarded to a gateway by publishing the relevant control event to an event service to which the gateway is connected. A request may be propagated over many gateways and event services in this fashion to reach a particular event service. In addition control events may be passed to a gateway by other means such as user input or via command line parameters.

An event service that a gateway, or any event service client, is directly connected to is known as a *direct event service*. An event service that a gateway, or any event service client, is not directly connected to is known as an *indirect event service.* The terms local and remote are not used, as process-wise gateway may be remotely or locally connected to an event service. An event service that is used to route a request is known as an *intermediate* event service.

5.2 The FES event model

This section presents the FES event model. The event model acts as a common language between event services. The FES design uses a two-step translation process to translate requests between event services. The first step involves translating the source event service request to a FES request. The second step involves translating the FES request to a destination event service request. FES adapters are required to translate event service specific requests to and from FES specific request. New kinds of event services may be added without affecting the existing system or existing adapters. This is how the FES supports multilateral heterogeneous event service inter-working.

The basic types section defines a useful set of primitive types. The event section defines the FES structured event type. The propagation model and filtering model of the FES are discussed in the next two sections. Finally the requests that are support by the FES are defined.

5.2.1 Basic Types

Good basic type support reduces the work that applications have to perform when mapping to an event model and reduces the chances of error and ambiguity. However increasing basic type support increases the size and mapping complexity of FES components. This overhead may not be required for a lot of applications. The FES event model defines the following set of basic types. This is a useful set that is sufficient to capture the basic type needs of the FES for proof of concept. These basic types are based on CORBA basic IDL types as described in [14].

	Description
string	An unbounded, null terminated string composed of ISO Latin-1 characters.
double	IEEE double precision floating point number.
long	Integer, range -2^{31} to $2^{31}-1$.

5.2.2 Event

A FES event is a structured event that is composed of a subject, a set of parameters and a set of attributes, as shown in Fig. 5.2. The FES allows parameter and attribute access by index as well as by identifier. Identifiers are case sensitive.

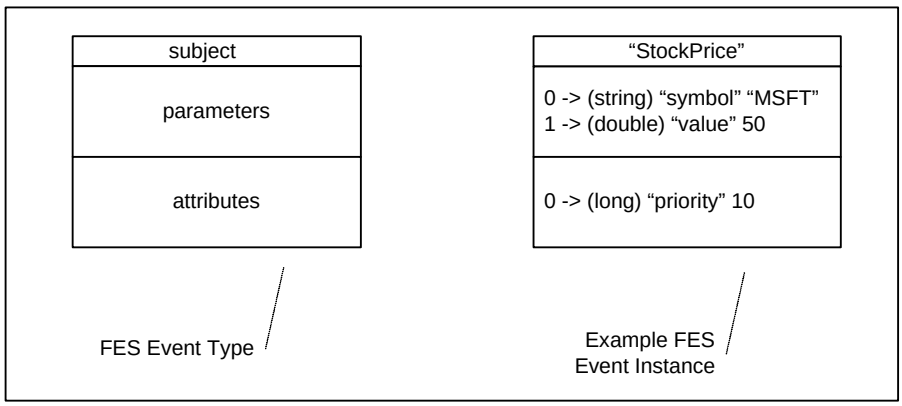

Fig 5.2 FES Event type and example instance

The structured event type was chosen as this type is commonly supported in event models. It allows flexible filtering. It is relatively easy to map an un-typed event to a structured event. The CNS specification [8] defines how CES un-typed events should be mapped to CNS structured events. The CES/CNS typed events are rarely used, as they are difficult to understand and implement [8, pp212]. STEAM and CNS structured events allow parameter access by index. Parameter access by identifier is necessary to support event services such as SIENA that do not provide parameter access by index. Identifiers are case sensitive, as this requirement will support both case sensitive and case-insensitive event services.

Subject

The subject identifies the application event type, e.g. "DiskFull", "DeviceOffline". It is of type `string`. To distinguish between events in a FES System the user must provide a unique event subject. The subject is case-sensitive.

Parameters

Parameters specify event data. An event may contain 0 or more parameters. A parameter consists of an event unique parameter index of type `long`; an event unique

parameter identifier of type `string`; a type identifier of type `long` that specifies the type of the parameter (can be `string(1)`, `long(2)`, `double(3)` or `Event(4)`); and the parameter data. Note: Events may contain other events. This is a useful feature that is used by control events.

Attributes

Attributes, as explained in the overview chapter, contain information pertaining to the context in which an event is generated such as priority, location, proximity, etc. Attributes have the same structure as parameters.

5.2.3 Attribute Support

The FES supports any event service attribute that may be applied on an event-service-by-event-service basis (hop-by-hop) by adapters. Adapters may ignore attributes that they do not recognise or attributes that their event services do not support. This allows support for new features to be readily added in the future without breaking existing code. The potential set of features is open-ended. Here is an example set of attributes.

Attribute	**Event Delivery Priority**
Attribute Identifier	Priority
Attribute Type	`long`
Description	
This attribute is based on the QoS priority attributes as supported by CNS. This attribute defines the order in which events are delivered to a consumer. Range is -32767 for lowest priority to 32767 for highest priority.	

Attribute	**Event Delivery Proximity**
Attribute Identifier	Proximity
Attribute Type	`string`
Description	
This attribute is based on the proximity attributes as provided by STEAM. The proximity attribute defines the range in which the event is valid. If an event has exceeded its proximity then it should be discarded.	

It consists of a proximity range reference point, specified as latitude and longitude values of type `double`. The shape of the range is also specified. It may be circular in which case a radius is supplied or it may be rectangular in which case a longitude and latitude dimension is supplied. The shape is identified by a `long` value. 0 = Circular, 1 = Rectangular. Ranges may be absolute or relative to the specified range reference point. A `long` value is used to identify the type of range. 0 = absolute, 1 = relative.

For circular proximities the attribute value takes the form `"type, sub-type, latitude, longitude, radius"`. For rectangular proximities the attribute value takes the form `"type, sub-type, latitude, longitude, dimx, dimy"` where:

type	= 0 for circular shape and 1 for rectangular shape.
sub-type	= 0 for absolute proximity, 1 for relative proximity
latitude,longitude	= reference point for proximity (origin).
radius	= radius of a circular shape.
dimx,dimy	= dimension of a rectangular shape.

5.2.4 Propagation Model

The adapter must push events to the gateway. For event services that don't support push, the adapter will have to pull events from the event service in a separate thread and then push these events to the gateway.

5.2.5 Filtering Model

The FES supports event filtering via the FES filtering language. At a minimum the FES filtering language must support subject based filtering. Subject based filtering is enough filtering support for the FES proof of concept. STEAM, COSMIC and CNS have the concept of a subject and provide subject-based filtering. SIENA mapping can easily define one of its parameters as a subject. The approach to filtering taken by the FES as described here does not depend on the extent of it filtering prowess and adding support to the FES for every conceivable filtering requirements dilutes the thesis of this project.

The FES makes use of two filters whenever a consumer makes a subscription request to an indirect event service: the subscription as made by the consumer at the direct event service in the direct event service filtering language (*direct filter*) and the subscription as made by a gateway on behalf of the consumer at an indirect event service in the indirect event service filtering language (*indirect filter*).

Filtering information may be lost when mapping to and from the FES filtering language. However the indirect filter must always define the same set of events or a superset of the events that was defined by the direct filter. In the case where a superset of events is specified at the indirect event service, unwanted events will cross the FES system to the direct event service. However these events will not reach the consumer, as the direct event service filter will filter them out.

The FES filtering language may be a perfect match for some event services, but may be less expressive/coarser for other event services. At all times a widening conversion may be applied but never a narrowing conversion. The original filter in the FES filtering language must be preserved at all times so that it may be applied consistently at all indirect event services.

5.2.6 Event Size

The FES does not place any limit on event size. This includes subject length, number of parameters, parameter name length, number of attributes, attribute name length, parameter value length and attribute value length. This version of the FES does not support event fragmentation.

5.2.7 Naming

The following elements in a FES system require identification. These identifiers are of type `string`. It is up to the user (or possibly a tool) to ensure that identifiers are unique. All of these identifiers are specified via FES configuration information.

	Description
FES System Identifier	Name is globally unique
Event Service Identifier	Name must be unique among event services in a FES System.
Event Subject	Name must be unique among events in a FES System. Case sensitive to support both case insensitive and case sensitive event services. Names with the prefix “FES_” are reserved for use by the FES.
FES Gateway Identifier	Name must be unique among gateways in a FES System.
FES Adapter Identifier	Name must be unique among gateways in a FES System.

Table 5.1 – FES Identifiers

Distribution Lists

A client of a FES system may specify the event services that a request is sent to. For example, a client may require that only a subset of the event services in the FES system receive announcements or subscription requests. This is an important requirement as it improves the scalability of the system. Otherwise requests would have to be propagated to all event services.

A distribution list provides functionality that is not part of normal event services, i.e. clients may target specific sets of subscribers and publishers. However clients need not be bound to certain event service instances. Instead they may specify the type of event service, or a range of event services to send a request to. This requirement helps to preserve the advantages that decoupling provides in event services. It is up to the application developer to name event service instances appropriately to the application in hand.

A distribution list is a comma-separated string composed of event service identifiers. The wildcard characters '*' and ‘?’ are supported.

Examples:

There are 10 event services in a FES system, ES1 to ES10.

The distribution list:

"ES1, ES2, ES3" specifies ES1, ES2 and ES3

"ES*" specifies ES1 to ES10.

“ES?” specifies ES1 to ES9.

“*” specifies all event services.

“” specifies no event services.

5.2.8 Requests

Requests define the functions that are supported by the FES. All requests except publication requests emanate from a single event service client. Publication requests are generated by the FES on receipt of events from event services. Requests may be distributed to one or more destination event services by the FES. The naming section defines how destination event services may be identified. The FES does not return the status of requests back to the caller. Requests are one-way functions that may be applied at most once to each destination event service. The following fundamental event service requests are supported by the FES.

Announcement Request

An announcement requests specifies a particular event type that may be published by an event service producer. Event services may propagate this information to consumers. This facility allows event services and consumers to prepare for future event arrival. The request takes a FES `Event` as a parameter.

Un-Announcement Request

A un-announcement request specifies an event type that will no longer be published by an event service producer in the future. This facility allows event services and consumers to tear down resources that are will no longer be required to handle events of a certain type. The request takes a FES `Event` as a parameter.

Subscription Request

A subscription request defines the events that a consumer of an event service is interested in. The consumer supplies a filter to specify this. The FES filtering model is described above. The request takes a FES `string` as a parameter.

Un-Subscription Request

A un-subscription request defines the events that a consumer of an event service is no longer interested in. The consumer supplies a filter to specify this. The FES filtering model is described above. The request takes a FES `string` as a parameter.

Publication Requests

A publication request defines an event that a producer of an event service has published to an event service. These requests are different from the other kinds of requests as they are generated automatically by the FES whenever it receives an event from an event service. The request takes a FES `Event` as a parameter.

5.3 FES gateways

The FES is realized by a set of event services that are connected by gateways. This section describes the gateway protocol in detail, including the various ways of issuing a request to the FES.

5.3.1 Control Events

As described in the overview section, gateways use asynchronous control events as a communication mechanism. Gateways subscribe to their direct event services (via adapters) for control events. Fig 5.3 shows some control event examples. A control event is a FES event with the subject “FES_ControlEvent” and with the following parameters:

Identifier	Type	Description
Type	`long`	Indicates the type of request. Announcement = 1, Subscription = 2, Publication = 3, Un-subscription = 4, Un-announcement = 5.
DistList	`string`	Specifies the distribution list of event services where the request should be applied.
Source	`string`	The identifier of the event service from which the request was made.
Filter	`string`	For subscription requests this parameter contains the subscription filter.
Event	`Event`	For announcements, un-announcements and publications this parameters contains a serialised FES event.

Table 6.1 – Control Event Parameters

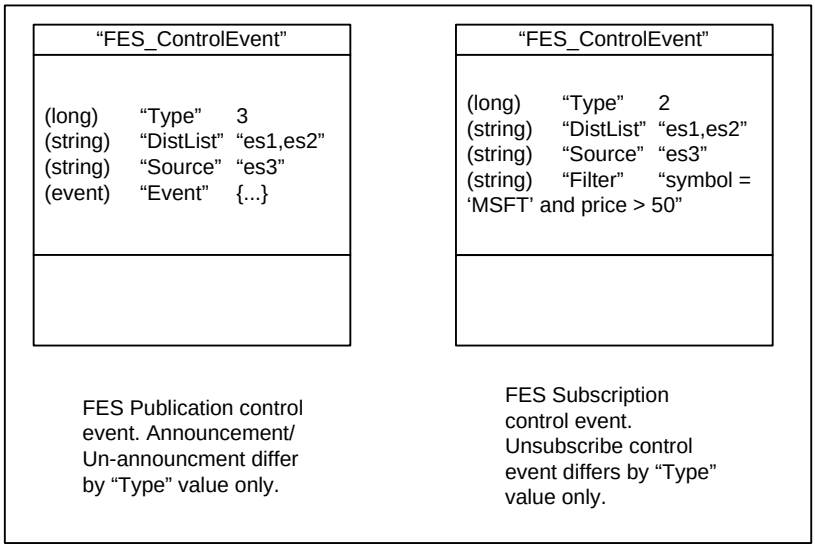

Fig 5.3 FES Control event examples

When an adapter receives an event it packages it up into a publication request and forwards the request (push) in a control event to its gateway for processing.

5.3.2 The Gateway Protocol

When a gateway receives a control event it examines the event's distribution list to determine whether the request contained within should be applied at a direct event service and/or whether the event should be forwarded to other gateway(s) for application at indirect event service(s).

If the request should be applied at a direct event service then the gateway unwraps the request details and carries out the necessary request. For example if the request is a subscription request, then the control event contains a filter. The subscription request is then made via the event service's adapter. If for example the request is a publication request then the control event contains an event. This event is extracted and published to the event service via the adapter.

If the request should be applied at an indirect event service(s) then the gateway must make a routing decision to decide which of its directly connected event services it should publish the control event to in order to route the request to the correct gateway(s). Please see the Routing section below for more details on this.

The gateway must manage some local state information regarding the requests that it has made to its direct event services. This includes information pertaining to the events that have been announced at a direct event service. This allows the gateway to announce event types when necessary, before publishing events of that type. This is necessary, as some event services don't provide event announcements while others do. This information also includes the subscription filters that were applied at each direct event service and the source of the subscription request. When a publish request is then received from an adapter, the gateway can determine the distribution list for the event. The gateway must examine the event to determine which filter it applies to, and thus determine which event service issued the original request. Subject based filtering simplifies this event–filter-matching process.

5.3.3 Example

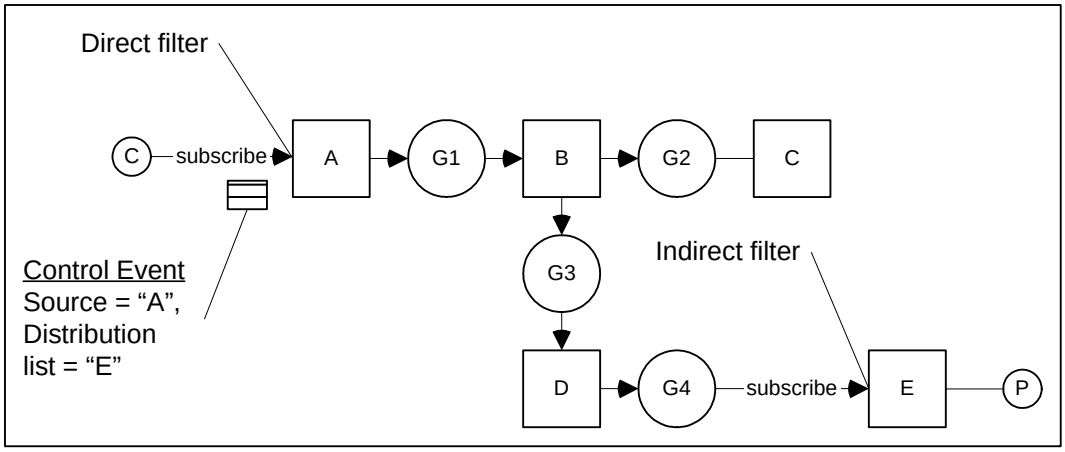

Fig 5.4 FES Gateway protocol – opaque subscription request

Fig 5.4 shows how a subscription request is routed from a consumer at event service A to event service E over three intermediate event services and three gateways. Note: It is assumed that an intelligent routing algorithm is used (not flooding).

The consumer issues a subscription request to event service A through event service A’s interface. The consumer then issues the same subscription request to event service E from event service A by publishing a control event to event service A. When G1 receives the control event, it extracts the distribution list to determine that the request should be applied to event service E. It therefore forwards the control event by publishing it to event service B. Likewise Gateway G3 then receives this control event and publishes it to event service D. Finally gateway G4 receives the control event, extracts the filter and makes a subscription request to event service E after making a note of the filter and the source of the request.

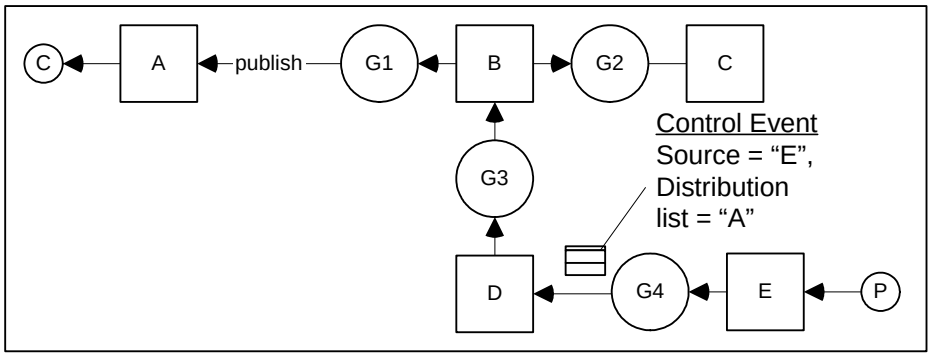

Fig 5.5 FES Gateway protocol – Publication request

If G4 should subsequently receive an event from event service E for that subscription, it then encapsulates that event in a publication request with the distribution list calculated as event service A. This request is sent by publishing it as a control event to event service D. As shown in Fig. 5.5 the request follows the reverse path of the original subscription request. G1 finally receives the request and applies it to event service A, i.e. it extracts the event and publishes it to event service A. The consumer receives the event since it has made a relevant subscription directly to event service A.

5.3.4 Transparent Requests

As shown above, publication requests are handled transparently by the FES. In addition announcement, un-announcement, subscription, and un-subscription requests may be injected into a FES system at a gateway. This kind of interaction is transparent to existing event systems in the federation. Transparent requests have a number of advantages. Existing event systems do not need to be touched. Events may be mapped to event types that existing clients expect. It is also very suited to FES systems with static routing and filtering requirements. This might be a common situation for FES systems that contain real time event services. Figure 5.6 shows a transparent subscription request that is equivalent to the opaque request shown in 5.4.

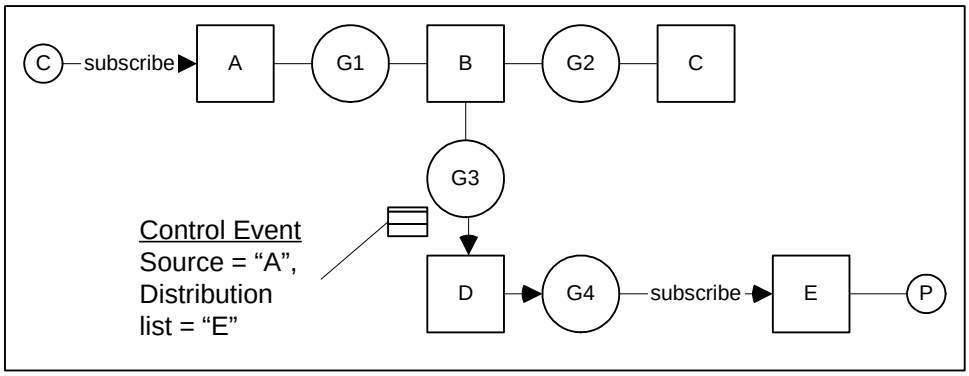

Fig 5.6 FES Gateway protocol – transparent subscription request

5.3.5 Opaque Requests

Many uses of the FES will require that event system clients are aware of the FES. For example to propagate event service client announcements and subscriptions as they occur to the FES, the client must inject the relevant control event into the event service. This is necessary as event services do not generally expose these requests and certainly do not support FES concepts such as distribution lists. Since the event service adapter already knows how to map to and from the FES event model, its code should be reused to perform the necessary conversion at the event service client. Fig 5.4 shows an example of an opaque subscription request.

5.3.6 Routing

This version of the FES does not define how routing is performed. Flooding could be employed as a first step – i.e. control events are output on all lines except the line that they arrived on. Quite elaborate routing mechanisms and optimisations could be employed based on the requests and events received at a gateway. For a good introduction to routing issues in computer networks see [13].

The user defines the topology of a FES system via the FES system configuration (see below). It is up to the user to ensure that there are no loops in this topology, i.e. the topology defines an acyclic graph.

The system developer must be aware of the fact that STEAM entities are mobile, are frequently out of range and could interface to a fixed event service from many points.

For example, with the current design, it may not always be possible to route a subscription request to a STEAM client. One way around this problem is to inject a subscription request into a gateway located with each STEAM client. When the mobile STEAM client/gateway comes within range of a fixed STEAM client/gateway, events may be propagated as normal to the fixed network. The demo application as described in the implementation chapter takes this approach.

5.4 FES adapters

The adapter pattern [12] is used to encapsulate heterogeneity among event services in the FES. This includes encapsulating event service requests and the mapping of FES requests to event service specific requests and vice-versa. This has a number of advantages. It is easy to add support for new kinds of event services and new kinds of event mapping functionality in the future without breaking existing systems. There is a danger that a badly designed adapter interface may impose too much work on implementers. Implementers may then decide not to use the FES and to roll their own inter-working solutions. Therefore this design aims to minimize the complexity, work and responsibility of an adapter.

5.4.1 Event Mapping

The FES design does not mandate how adapters perform event mapping. However the mapping support provided by adapters may be classified as follows:

User-defined event mapping

With user defined event mapping the adapter allows the user (system developer or administrator) to define how events are mapped from an event service event model to the FES event model and vice-versa. The user could supply this information on an event basis via configuration file(s). The adapter would then process this information whenever an event of a certain type was received to map to the FES event model. The user could also supply plug-in code to map events of a specific type. The adapter would load this code to process certain events (as done in [11]). User-defined event mapping can give the user complete control over the event mapping process.

Automatic event mapping

Here the adapter automatically processes and maps events it receives from its event service to FES events. The FES event type readily facilitates automatic event mapping

since this type is a structured event that can be queried at run time for parameter, attribute, and subject identifiers, types and values. This approach can work well for the structured event type. However this cannot be used for un-typed event mapping (e.g. CES, COSMIC), or where event services cannot support the size of the event to be mapped. Automatic event mapping can cater for such structured event events types where they are created dynamically at run time. Considering the number of event services that may be involved, the rules for automatic event mapping must be simple and well-understood otherwise subscribers will not know what kind of events to expect.

Combined automatic and user-defined event mapping

The adapter can implement a combination of automatic and user-defined event mapping. This is a flexible approach. It allows the adapter to automate as much of the event mapping process as possible, with the user supplying the necessary mapping information where required. It also allows the user to over-ride automatic event mapping if necessary, for example where automatic event mapping would not generate the ‘right’ event structure with respect to an existing application.

Control event mapping

It is up to adapters to decide how to represent control events on their event service. The integrity of a control event must be maintained at all times so that requests may be applied consistently at event services. Control event size can vary dynamically since they may contain serialized FES events. Therefore depending on the maximum event size in a FES system, event services with limited event size may not be suitable as intermediate event services. It is the responsibility of the application developer to ensure that application requests can propagate as control events across all intermediate event services in a FES system.

5.5 Configuration

Configuration information is required to identify the event services instances, event service types and gateways in a FES system. It specifies the event services which each gateway bridges. Where user-defined mapping is used it specifies the mapping between FES events and event service events. This information could be provided by many means including a configuration file (using XML perhaps), command line parameters, a user interface, or hard-wired into a gateway. A tool could be employed to manage configuration information, especially for large FES systems or where a large amount of event mapping is required. The information could be manually deployed at each gateway. A future enhancement could provide for the distribution of this information automatically to all gateways via the FES.

At a minimum a FES system requires the following configuration information:

Event Services - An event service is an instance of a particular event service type and version. To distinguish between event service instances the user must provide a unique event system instance identifier and event service type identifier. Each instance may have a set of parameters that are required to connect to it and/or to configure the connection. These parameters are event service type specific.

Gateways - A gateway bridges two or more event service instances. The user must specify these instances by listing their unique event service identifiers. To distinguish between gateways the user must provide a unique gateway identifier. Based on the event service type the gateway can locate and load the relevant adapter. For example the event service type could identify a particular component that implements the adapter interface.

5.6 Scalability

The FES addresses scalability issues as follows:

- It is a federation of event services. There is no central point of control or failure. There is no global state to manage.
- Distribution lists allow targeting of a subset of event services in a FES system
- Subscription filters across event services are supported. This reduces the amount of events that potentially would have to cross the federation.
- A scalable event service such as SIENA may be used to create a large geographical scale FES event system.

5.7 Request reliability and error handling

This version of the FES does not facilitate the return of request status information to the caller. This is difficult to achieve for the following reasons: requests may propagate across a wireless or a ad-hoc wireless network with little guarantee of a return path; event services, many of which have best effort event delivery semantics are used to propagate requests; events services within the federation may be down at any time; a request may propagate successfully to many event services and not successfully to other event services. The FES offers ‘at most once per event service’ delivery semantics, i.e. the request will be delivered to each event service in the distribution list at most once.

At a minimum each gateway should log diagnostic information and error information to help locate the source of an error.

5.8 Use Case – Traffic Monitoring System

5.8.1 Overview

A use case is presented here to illustrate and test the design of the FES. This use case describes a traffic monitoring system, set in the not too distant future. This system monitors traffic speeds at various locations in a city and logs the license number and speed of cars that exceed speed limits. Offending drivers are automatically issued with speeding fines. The system is outlined in Fig. 5.7.

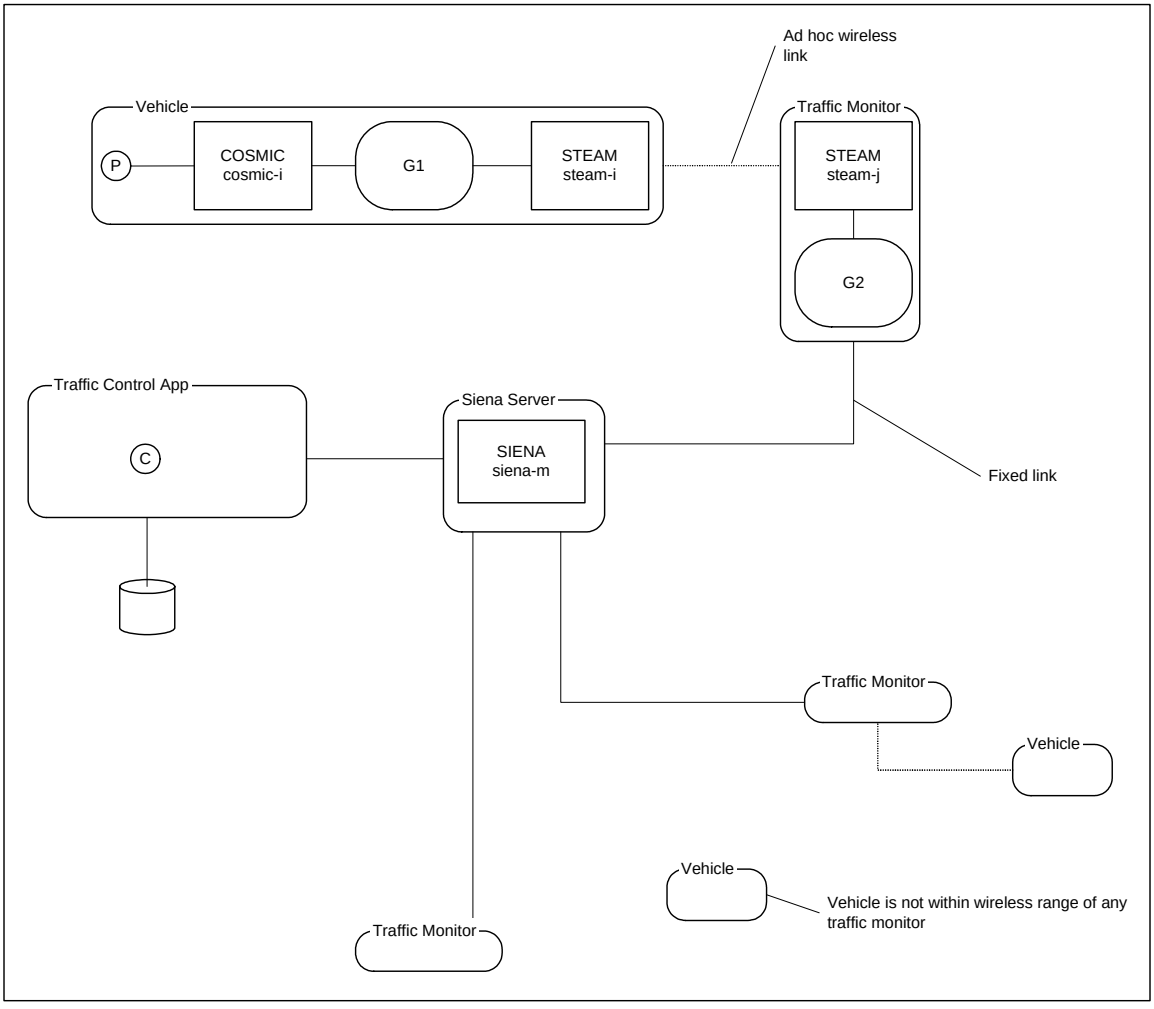

Fig 5.7 Traffic Monitoring System

The system assumes that all cars (due to regulations) are fitted with tamperproof components that broadcast various events over an ad hoc wireless STEAM event system. These events include current location, current speed, breaking status and so on. The current speed of the car is published every second on the car's onboard real time network via a COSMIC event system. A "Speed" event is used to contain the car's current speed and its license number. A FES gateway is used to inter-work the COSMIC and STEAM event services.

Fixed roadside traffic monitors located at or near speed limit signs subscribe for these speed events and publish them on a wide area fixed SIENA event service. Each monitor contains a FES gateway inter-working the STEAM and SIENA event services. The subscription filter employed at the STEAM event service in each monitor depends on the speed limit in the area. For example "subject = 'Speed' and rate > 30" would apply in a 30mph area.

In the city traffic control office there exists a traffic control application. This application allows the operator to set the speed limits for various areas in the city. The roadside signs dynamically display the current speed limit. In addition, setting a speed limit changes the corresponding subscription to the STEAM event service at the roadside monitor.

The following notation will be used to identify event service requests and control events.

	Meaning
s(f)	Subscribe for events with filter f.
us(f)	Un-subscribe for events with filter f
p(x)	Publish event x
c(es,x,d)	Control event encapsulating event service request x to be applied at event service instance(s) specified in distribution list d. Request originated from event service es. For example: c(k,s(f),{i,j}) = subscribe for events with filter f at event services i and j. Request originated from event service k.

5.8.2 System configuration

There exists in each car a COSMIC producer producing “Speed” events on the COSMIC event service. The traffic control application in the city traffic control office subscribes to SIENA for all “Speed” events on start-up.

A FES event is required to represent the COSMIC speed event. The FES “Speed” event is quite simple, consisting of a “license_no” parameter of type `string` and a “rate” parameter of type `double`. This is a simple event type and no user-defined mapping information is required for the STEAM and SIENA adapters to handle events of this type. The event will be delivered on a best effort basis over STEAM and SIENA. The event will be propagated over STEAM with the default proximity of absolute circular proximity (centred at the car) of maximum wireless range.

The COSMIC adapter does require user-defined mapping information. For this example: COSMIC event id 12345 represents a “Speed” event. The license number is contained in the first 7 data bytes of this event and the speed of the car is contained in the last data byte. Therefore the following parameter mapping information could exist for COSMIC to FES conversion: {12345 ⇔ subject ”Speed”, 0..6 ⇔ parm. 0 “license_no” `string,`7 ⇔ parm 1 “rate” `double}`. This mapping information could be statically supplied by the car control system.

The STEAM/COSMIC gateway in each car is configured with a FES subscription via the following control event: c(“steam-i”, s(“subject = Speed”), “cosmic-i”). This subscription is static – it is injected into the gateway by the car’s control system. On starting the gateway thus subscribes to the COSMIC event service cosmic-i for all speed events. The STEAM event service steam-i is the source of the subscription.

The event service identifiers in each car need not be unique in this example because the only output from the car that matters is the “Speed” event. In other use cases a way of ensuring uniqueness for car event service identifiers may be required (it may not be feasible to statically configure identifiers for each car) – perhaps by using the car license number in event service identifiers. The SIENA event service identifier, siena-m, is unique as there is only one instance in this example. The STEAM event

service identifier must be unique at each gateway – each STEAM access point is treated as a separate event service. Identifiers could be statically configured.

5.8.3 Setting a speed limit

The Traffic Monitor gateways are dynamically configured with subscription requests that are sent from the traffic control application over the SIENA event service as SIENA control events.

When the controller changes a speed limit in an area, an unsubscribe SIENA control event c(“siena-m”, us(“subject = Speed and rate > s1”), i) followed by a subscribe SIENA control event c(“siena-m”, s(“subject = Speed and rate > s2”, i) are injected into the SIENA event service by the traffic control application, where s1 represents the old speed limit, s2 represents the new speed limit and i is the identifier of the STEAM event service where the subscription should be made.

If the operator wished to set the speed limits of a set of traffic monitors then a wildcard or a list of event STEAM event services could be specified in the control event distribution list.

5.8.4 Catching offending drivers

When a car’s speed is published on the COSMIC event service the COSMIC adapter receives the event and maps it to a FES event, p(x), via the supplied “Speed” mapping information. This event is forwarded as a FES publish control event to the gateway in the car. This gateway announces and publishes the event to the STEAM event service via the STEAM adapter. The STEAM adapter automatically maps the announcement and publication to STEAM specific requests.

If the car is within range of any traffic monitor, then the STEAM event service applies its proximity and content filtering. If the car is travelling over the designed speed limit then the STEAM adapter in the traffic monitor gateway receives the event. It then maps this event into a FES publish control event, and forwards it to the gateway. The gateway announces and publishes the event on the SIENA event service via the SIENA adapter.

The traffic control app then receives the “Speed” event and logs the offending driver's details to a file.

5.8.5 Synopsis

This use case demonstrates the following features of the FES design:

- User-defined event parameter mapping to map restricted COSMIC events to FES events.
- Automatic event parameter mapping used elsewhere in the system.
- Event Service Transparency. All event services remain unmodified and unaware of the federation. The traffic control application is FES aware as it needs to inject control events into the federation each time a subscription changes. There is a trade-off here: filtering events closer to the source and some loss of transparency vs. filtering events at destination and complete transparency.
- Dynamic subscription routing across an event service. Subscriptions are routed by the FES to the appropriate event services (SIENA to STEAM).
- Event routing across a real time event service and a mobile location aware event service to a fixed event service. QoS features are not covered here and all work is best effort. Proximities are defaulted to a max. circular wireless range.

The implementation of the FES includes a demonstration/test application that implements this use case. This is described in the next chapter.

5.9 Gateway Interface

The section defines the interface of a FES gateway. A simple CORBA IDL like syntax is used. The `forward` method provides a more precise statement of the gateway protocol as defined in the FES gateways section.

5.9.1 start

```
void start(
          [in] string    gatewayId,
          [in] string    configInfo
     )
```

Loads the adapters as specified in the configuration information. Connect these adapters to their event services by calling their connect methods.

Parameters

`gatewayId`	Gateway identifier.
`configInfo`	Configuration information. At a minimum the configuration information should specify the adapters that must be loaded, specify their identifiers and specify the configuration information for those adapters. It is up to the implementation to determine what other configuration information is required and how it is accessed. For example this parameter may contain a reference to an XML file that contains configuration information.

Throws

`Exception` if an error occurs.

5.9.2 stop

```
void stop()
```

Disconnect all adapters from their event services via their disconnect methods. Unload all adapters.

Throws

`Exception` if an error occurs.

5.9.3 forward

```
void forward(ControlEvent e, string adapterId)
```

If control event e contains a publication request and the distribution list is empty then determine the distribution list – it is the set of original subscribing event services. Update e with this distribution list.

For all the adapters in this gateway (except the adapter from which this source event came (i.e. adapterId)), check control event e's distribution list to see if the request is applicable to the adapter's event service.

If the request is applicable to the adapter's event service
Then

If this is an announcement request and the event type has not been announced before at this event service
Then

Announce the event via the adapter announce method. Make a note of the fact that the event has been announced at this event service by the source event service.

End If

If this is a un-announcement request and the event type has been announced at this event service before by the source event service
Then

Un-announce the event via the adapter un-announce method. Remove the announcement note.

End If

If this is a subscription request

Then

Make a note of the subscription filter and request source so that received events may be routed back to the correct subscribing event services. Note: it is straightforward to map received events to filters for simple subject filtering. More complex filtering may be more difficult to handle.

Make a subscription request via the adapter subscribe method.

End If

If this is an un-subscription request

Then

Make a un-subscription request via the adapter subscribe method. Remove the subscription node.

End If

If this is a publish request

Then

If this event type has not been announced before at this event service then announce the event via the adapter unannounce method.

Publish the event via the adapter publish method.

End If

End If

If the event is applicable to other event services then forward the control event by passing it to the adapter's publish method (flooding used).

Parameters

`e` A control event.

`adapterId` Identifier of directly connected adapter .

Throws

`Exception` if an error occurs.

5.10 Adapter Interface

To support request mapping adapters must implement the following interface:

5.10.1 connect

```
void connect(
                [in] Gateway   g,
                [in] string    eventServiceId,
                [in] string    adapterId,
                [in] string    connectionParms
           )
```

Connect to the event service using the supplied connection parameters (connectionParms). This is a logical connection request only, required for the adapter to set up the necessary resources to accept further requests for the event service. Event services may not have the concept of a connection.

If the event service requires announcements then announce that this adapter intends to publish control events to the event service. Subscribe to event service for control events where the source of the event is not this adapter (remember this adapter is also publishing events to the event service).

All received control events should be passed to the gateway via the gateway `forward` function for processing. All other events received should be converted to publication control events and forwarded to the gateway.

Parameters

`g`	This adapter's gateway.
`eventServiceId`	The identifier of the event service that this adapter is connect to.

`adapterId`	The identifier of this adapter.
`connectionParams`	Opaque string, containing adapter and event service specific connection and configuration information.

Throws

`Exception` if an error occurs.

5.10.2 disconnect

```
void disconnect()
```

Disconnect from the event service. Tear down any resources if necessary (e.g. unannounce, unsubscribe).

Throws

`Exception` if an error occurs.

5.10.3 getId

```
string getId ()
```

Returns this adapters identifier.

5.10.4 getEsId

```
string getEsId ()
```

Returns this adapters event service identifier.

5.10.5 announce

```
void announce([in] Event e)
```

Maps event to the event service event model. Announce event. If the event service does not support announcements then do nothing.

Parameters

`e`	The event to announce.

Throws

`Exception` if an error occurs.

5.10.6 unannounce

```
void unannounce([in] Event e)
```

Maps event to the event service event model. Un-announce event. If the event service does not support announcements then do nothing.

Parameters

`e`	The event to un-announce.

Throws

`Exception` if an error occurs.

5.10.7 publish

```
void publish(Event e)
```

Maps event to the event service event model. Publish event.

Parameters

`e`	The event to publish.

Throws

`Exception` if an error occurs.

5.10.8 subscribe

```
void subscribe([in] string filter)
```

Map filter to event service filtering language. The mapped filter must specify the same set or a superset of the events as specified in the original filter.

Make a subscription to the event service with the mapped filter.

If the event service does not support subscriptions then do nothing.

Parameters

`filter`	Filter in FES filtering language.

Throws

`Exception` if an error occurs.

5.10.9 unsubscribe

```
void unsubscribe([in] string filter)
```

Map filter to event service filtering language. The mapped filter must specify the same set or a superset of the events as specified in the original filter.

Make a un-subscription to the event service with the mapped filter.

If the event service does not support subscriptions then do nothing.

Parameters

`filter`	Filter in FES filtering language.

Throws

`Exception` if an error occurs.

Chapter 6. Implementation

This chapter describes an implementation of the FES. The aim of this implementation is to show that the FES design is coherent and viable. The implementation includes a generic gateway component and an adapter for the SIENA, STEAM and CNS event services. An implementation of the use case presented in the previous chapter, produced as proof of concept, is also described here.

6.1 Possible approaches

Two approaches were considered for implementation of the FES:

6.1.1 Compiled Approach

With this approach, a configuration file that describes event, event mappings, event services and gateways is input into a tool that generates FES systems. This tool produces the necessary FES system code including gateways and adapters. It should be possible to plug in support for different types of event services to the tool without affecting the existing tool code. The configuration file could be created manually or via some other tool. Therefore implementing the FES involves implementing this tool and the necessary tool plug-ins for participating event services.

This approach will produce efficient run-time translation and mapping code, as there is no need to look up and interpret this information at run time. This approach can also produce closer mappings to event service APIs and interface languages. However a change in the configuration will require a re-build of the system or parts of the system and a reinstallation.

6.1.2 Interpreted Approach

This approach requires the development of a generic gateway component and an adapter for each event service. Gateways and adapters read event mapping and configuration information on startup and apply this information when translating and mapping data. Therefore implementing the FES involves implementing a generic gateway component and an adapter for each participating event services.

This method produces slower run-time code than the compiled approach as configuration information is accessed and interpreted at run time for each event and request. However a change in the configuration will only require a restart of the relevant FES components.

It was decided to implement the interpreted approach. This method is easier to develop, test and debug. A future enhancement would provide the compiled approach as an optimization enhancement.

6.2 Overview

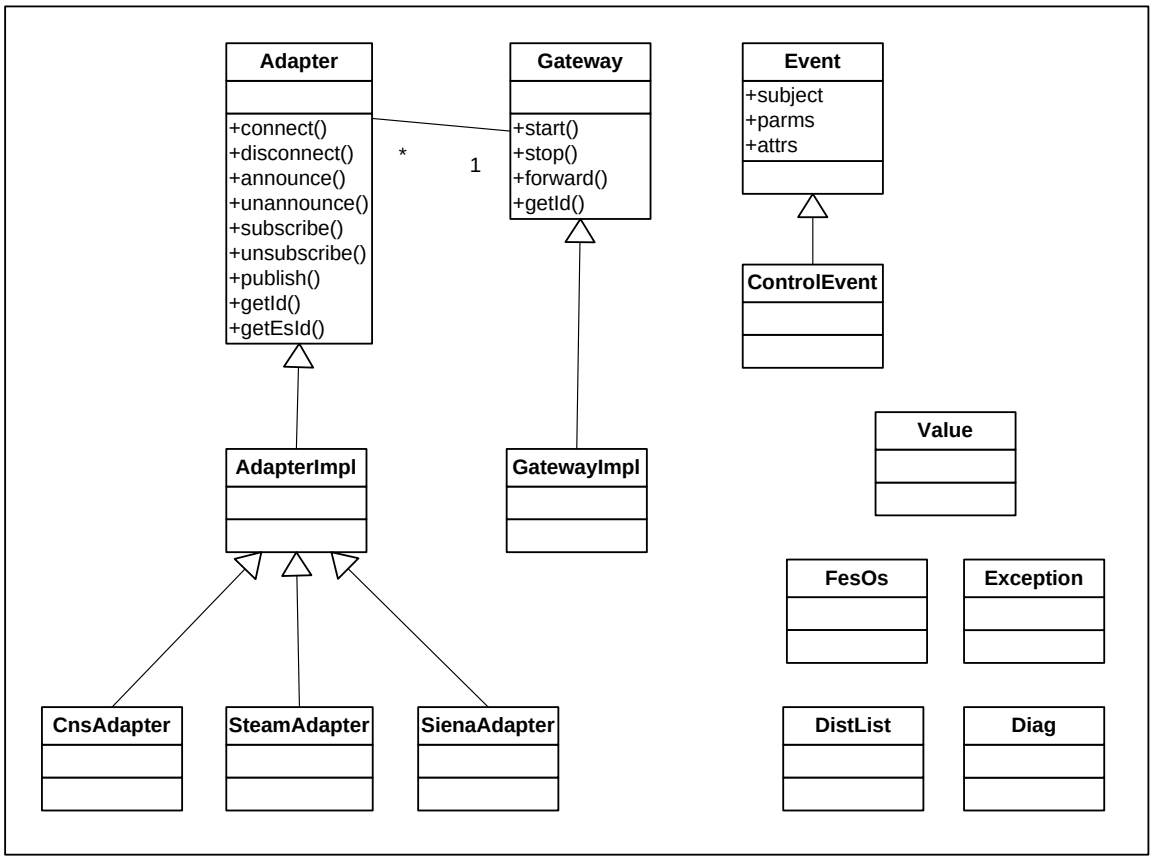

Fig 6.1 FES Implementation – UML Class Diagram

To test the FES design and the traffic monitoring use case sufficiently, the STEAM, SIENA and CNS event services were chosen as FES participants. CNS was used instead of COSMIC as the COSMIC event service was not available at implementation time. CNS provides a standard event service and QoS capabilities. As shown in the Issues chapter, these event services have sufficiently different event

models, event services, feature sets and implementations to test the FES design sufficiently. The development platform was Visual C++ 6.0 on Windows 2000 Professional [26]. The Win32 TAO CNS implementation was used [23].

Fig 6.1 shows a UML [24] class diagram of the implementation. One aim of the implementation was to keep it as portable as possible. To this end standard C++ [25] was used as the implementation language, with the class `FesOs` used to wrap operating system specific details. A CNS adapter, a SIENA adapter and a STEAM adapter were implemented via the classes `CnsAdapter`, `SienaAdapter` and `SteamAdapter` respectively. `AdapterImpl` is a base class that provides common adapter functionality such as consistent diagnostic logging and error handling.

The `Event` class encapsulates an event in the implementation. It contains the event subject (standard C++ `string`), a parameter map and an attribute map. The maps are standard C++ library `map`s that map `string` identifiers to a `Value` type. `Value` is a union type that can contain a `string`, a `long`, a `double` or an `Event` (events may contain other events). It is similar to the CORBA type `any` [14]. This class also contains methods to serialise and un-serialise events to and from a string. These methods are used to package an Event into an event service control event. `ControlEvent` inherits from `Event` and provides some helper functions for setting and getting control event parameters.

`DistList` encapsulates distribution list processing. `FesOs` wraps operating specific details such as loading dynamic link libraries , thread management, and thread critical sections. `Diag` provides diagnostic logging functionality. Finally `Exception` is a simple exception class for containing FES exception information.

The FES gateway is realised as a Win32 [26] executable that accepts configuration information via command line parameters. This information specifies the gateway identifier, the adapters to load, the adapter identifiers and the configuration information for each adapter. In addition, test GPS location information may be specified on the command line. This information is passed to the STEAM location service. For example the following command line in Fig 6.2 creates a gateway with

identifier “G3” at GPS position (0.032,0.004). The gateway has two adapters: a SIENA adapter with identifier “`Siena-2`” and a STEAM adapter with identifier “`Steam-3`”. “`localhost 9001 senp://localhost:6000`” is SIENA specific configuration information. “`20 20 0`” is STEAM specific configuration information.

```
Gateway G3 "0.032 0.004" "Siena Siena-2
localhost 9001 senp://localhost:6000 Steam
Steam-3 20 20 0"
```

Fig 6.2 Gateway Configuration

Each adapter implementation is realised as a Win32 dynamic link library that is loaded at run time by the gateway. Control events may also be passed to the gateway via command line user input. Fig 6.3 shows how these components may be arranged to address the traffic monitoring use case.

6.3 FES mapping

The use case event type is quite simple and all of the event services support structured events. Therefore the adapters need only implement automatic event mapping for proof of concept.

6.3.1 Basic Type Mapping

The following table specifies the mapping between FES basic event types and event service basic types.

	STEAM	SIENA (C++ API)	CNS
string	S_STR	std::string	string
long	S_INT	Int	long
double	S_DBL	Double	double

Table 6.2 – Event service basic type mappings

6.3.2 STEAM to FES Mapping

The range of the STEAM proximity discovery service, the default proximity type and the default proximity range may be passed to the STEAM adapter via its `connect` method. For testing, test GPS locations may also be passed to the STEAM adapter. STEAM provides a location service that may be configured with test data.

The subject filtering of STEAM easily maps to the FES filtering requirement.

The STEAM adapter publishes and receives control events. To distinguish control events that the adapter has produced from other control events, the adapter must put its identifier in the content of the event. Unfortunately, because content filtering is applied at the consumer in STEAM, this means that the STEAM adapter has to filter all of the control events that it is producing.

The STEAM structured event allows querying at run time for parameter type and value information. It is easy to therefore map between FES events and STEAM events at run time. STEAM proximity information is mapped to the FES “Proximity” attribute as described in the design chapter. The defaults passed to the `connect` method are used if this attribute is not specified.

STEAM separates event types, event data and proximity information into separated classes. Producers and consumers also use separate classes to access an event. This makes mapping to and from FES events more awkward than it could be.

STEAM does not support variable length string parameters. STEAM event type must specify the exact type and number of parameters. However control event data can vary in length. Therefore a STEAM control event consists of a fixed maximum number of string parameters. The control event data is serialised into the required number of parameters with the remaining parameters set to empty strings.

6.3.3 CNS to FES Mapping

The CNS adapter does not process any configuration information. It connects to a ‘hard coded’ event channel name. A basic improvement on this implementation would allow different event channels to be specified in the connection parameters.

CNS supports a powerful filtering language (trader constraint language) that allows filtering on any part of a CNS structured event. The FES event subject is mapped to the event_name field of a CNS structured event header.

The CNS structured event allows querying at run time for parameter and attribute type and value information. It is easy to therefore map between FES events and CNS structured events at run time. The adapter maps the priority attribute to the FES “Priority” attribute if present. It defaults it to 0 if not present (as described in the design chapter).

6.3.4 SIENA to FES Mapping

The SIENA adapter requires connection parameters that specify the SIENA server host and port plus connection parameters that specify the receiver host and port (required to receive events from SIENA). These are passed to the adapter’s connect method.

The SIENA C++ API does not support the event pull propagation model. Therefore the SIENA adapter manages a separate thread to pull events from SIENA and push these events to the gateway.

The SIENA structured event maps well to the FES event. However SIENA has no concept of an event subject. Therefore for automatic event mapping the SIENA parameter “FES_Subject” is used by the adapter implementation to specify the subject of the event. User-defined event mapping could allow other SIENA event parameters to be specified. SIENA filtering supports filtering on any parameter in the event.

The SIENA C++ API uses the ‘-‘ character as a delimiter when marshalling event parameter identifiers. Therefore this character cannot be used for event parameter identifiers in SIENA. This is probably a flaw with the SIENA C++ implementation. However this is the sort of real-world issue that a FES system has to deal with. Possible solutions include: (1) Do not use ‘-‘ in event parameter/attribute names when a SIENA system is participating in the FES. (2) Use the Java API. (3) A better solution might be to allow user configuration to specific a substitute character. This implementation uses option (1).

6.3.5 Other Issues

This implementation does not have to deal with event service language heterogeneity or event service platform heterogeneity since interfaces to the chosen event services are available for the same language (C++) and the same platform (Win32). A more comprehensive solution would employ CORBA interfaces to encapsulate this heterogeneity.

6.4 Test application

A test/demo application was developed to realise the traffic monitoring use case for proof of concept. Fig 6.3 outlines the components involved in this application. There are 8 processes involved: 3 gateways, a CNS publisher, a SIENA subscriber, a SIENA server, a CNS server and a CORBA name server [14]. The STEAM event service is collocated with the relevant gateways. 6 adapters are required – 2 per gateway. For the demo all processes were run on the same physical Win32 based machine.

Gateway’s G2 and G3 are fixed traffic monitors that inter-work SIENA and STEAM event services. In the ‘vehicle’ is a ‘mobile’ gateway G1 that inter-works CNS and STEAM event services. It is passed simulated GPS locations to ‘move’ it between traffic monitors. CNS PUB is a CNS publisher in the ‘vehicle’ that publishes “Speed” events every second to the CNS event service. This rate varies between 20 and 90 mph. SIENA SUB is a SIENA subscriber, subscribing for “Speed” events where the speed is > 40 mph.

A static subscription request is injected into G1 by the user to specify a filter of “Speed”, with a distribution list of “Cns” and with the source specified as “Siena”.

The gateway G1 then receives “Speed” control events from the CNS event service every second via its CNS adapter. The gateway determines the distribution list to be “Siena”. These control events are then mapped to and published to the STEAM event service every second via the gateway’s STEAM adapter. Whenever a traffic monitor is within range the gateway in the traffic monitor receives these events via its STEAM adapter. “Speed” events are extracted from the control events. These are then mapped to and propagated to the SIENA event service via the SIENA adapter. If the rate is > 40 mph then the subscriber SIENA SUB receives the event.

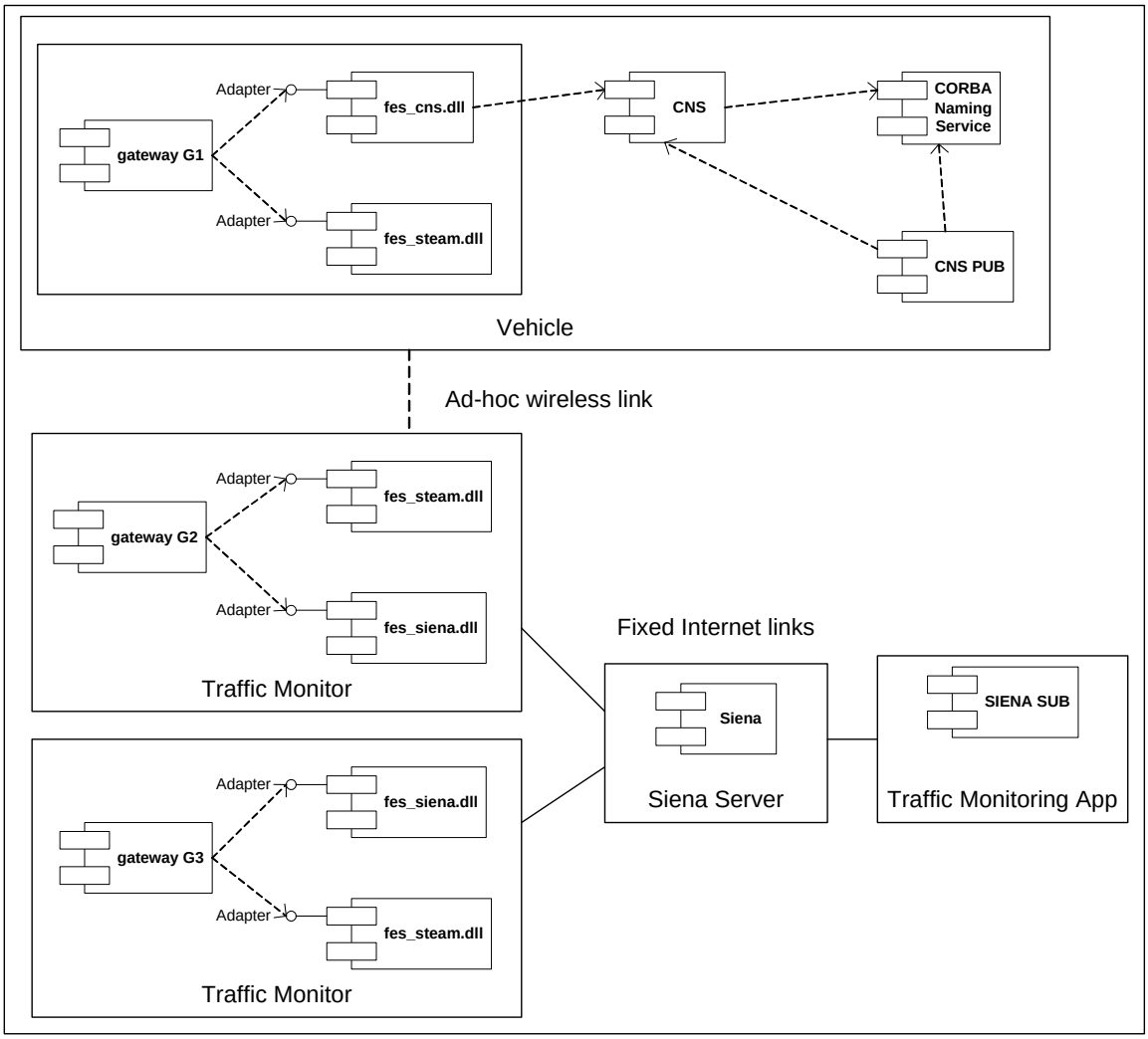

Fig 6.3 FES Implementation – Component Diagram – Traffic Monitoring Use Case

Each gateway, the CNS publisher and the SIENA subscriber log diagnostic information to separate log files (via the `Diag` class). This information may be used to verify that events and being propagated and mapped correctly. It may also be used for debugging purposes.

A log viewer application was written in Java, for viewing this information graphically. This is useful when multiple concurrent entities are involved and when entities have spatial relationships. This reads in the diagnostic information from the log files and sorts this information based on time (this is straightforward since the demo runs on a single machine). The user may assign an image to each entity in the system. The user may then step forward and back through the log information and the viewer displays the relevant information on screen. Fig 6.4 shows a screenshot of this application.

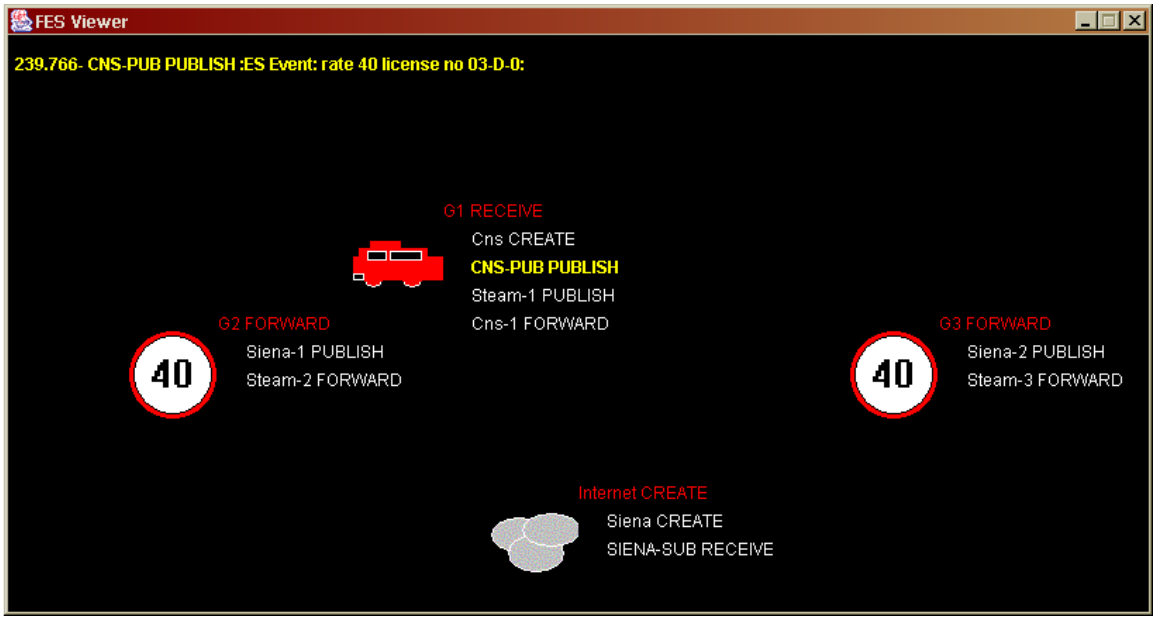

Fig 6.4 FES Implementation – FES Viewer application screenshot

Chapter 7. Evaluation

This chapter evaluates the FES design and implementation to determine the benefits of the solution for addressing the event service inter-working problem. The chapter also discusses the viability of federation in general for meeting certain kinds of system event requirements.

7.1 FES benefits

7.1.1 General solution

For the FES to truly claim that it addresses event service heterogeneity, it must support a wide variety of event models.

The FES can easily work with a mediator event model. An adapter may be connected to any mediator of a functionally equivalent mediator event model such as SIENA or an adapter may be connected to each mediator in a non-functionally equivalent mediator event model such as CNS. Adapters may be collocated with or separated from event service middleware. The FES can work with an implicit event model such as STEAM, by collocating an adapter and gateway with a STEAM client and by issuing subject based filtering requests to STEAM.

The FES requires the push event model. An adapter can easily convert a pull consumer event model to a push consumer as demonstrated by the SIENA adapter in the previous chapter. Other event service propagation models requirements could be handled in a similar fashion.

For parameter mapping requirements the FES solution is more than adequate. Parameters are just data that may be easily transformed and translated between event services by adapters. The FES works best with structured event types. This facilitates automatic event mapping. Un-typed events and typed events may also be catered for but this requires extra effort on behalf of the adapter implementer. The FES does not support parameter mapping for event types that are created dynamically at run time and whose parameters cannot be automatically mapped.

The FES supports subject based filtering. This is adequate for SIENA, STEAM and CNS. When direct and indirect filters do not match events may unnecessarily cross the system only to be filtered out at the direct event service. Some applications may not be able to afford this overhead. Therefore a stronger filtering language would benefit a general FES solution (e.g. the CNS trader constraint language). Another improvement would allow the support of multiple concurrent filtering languages. This would allow adapters to chose and apply the most descriptive filter at event services.

The FES approach works well for handling attribute requirements that can be encapsulated in events and processed on a hop-by-hop basis. For example, in the FES, the proximity semantics of STEAM can be encapsulated in an event attribute and applied on a hop-by-hop basis. All that is required is a location service at each adapter (such as STEAM's location service). Therefore events can be easily dropped if they exceed their proximity. The federation viability section below describes discusses this aspect further.

7.1.2 Low-risk solution

To allow an event service to participate in a FES system, the developer need only develop and test an event service adapter. The developer is exposed to the event service API only. There is no need to make any modifications to event services. Event service clients need only be modified when opaque requests are required. This is a decoupled development effort. The intellectual complexity of the system is reduced. The developer need not be an expert in event services and event service inter-working. Therefore event services may be adapted and tested quickly and cheaply.

7.1.3 Straightforward solution

The FES design is a straightforward solution. With a flooding and subject based filtering implementation, the gateway is quite simple to implement. There is very little burden placed on adapter implementers. The adapter interface is intended to be as complete and minimal as possible. By completeness, the interface aims to support a wide range of useful event service functionality. The interface is minimal in that there are few methods to implement, understand and there is no overlapping functionality.

These factors allow implementers to develop and test FES based solutions quickly and cheaply.

7.1.4 Flexible solution

The flexibility of the FES design gives developers a lot of choices when employing it to address event service inter-working problems. Since event mapping is hidden behind the adapter interface, system developers can employ whatever kind of mapping solution necessary to meet requirements. This could range from a simple adapter that is hard coded to handle a single event type to a combined user-defined and automatic event mapping solution.

The adapter approach allows systems to be composed dynamically and transparently at run time. This allows event services to be substituted for other event services, if necessary, without breaking the existing system. Event services could be upgraded to newer versions without breaking the existing system. For example, in the traffic monitoring use case, the SIENA event service could be replaced with a more reliable CNS event service if necessary. Only the traffic monitoring application would need to be modified – the rest of the system remains unchanged.

Finally the FES implementation may take the ‘interpreted’ or ‘compiled’ approach as explained in the implementation chapter. This allows better performing ‘compiled’ gateways to be substituted for ‘interpreted’ gateways if necessary.

7.2 Federation viability

This section discusses the viability of federation in general for meeting certain system event requirements.

Heterogeneous event service federation allows an event system to be dynamically composed of different kinds of event services. The event services may be chosen and connected so that they adequately address a particular event system requirement. Services may be selected in isolation, with the knowledge that the choice will not affect the rest of the federation. This transparency gives the system developer good scope when addressing requirements. For example, the FES test application offers a

realistic solution for inter-working ad hoc mobile event services such as STEAM with fixed event services such as SIENA and/or CNS. The choice of fixed network event service may be based on reliability (e.g. use CNS) or scalability requirements (e.g. use SIENA) and this decision is independent of the mobile event service choice.

Federation may also meet requirements through event service synergy. For example, STEAM/CNS inter-working provides a mechanism to extend the proximity capabilities of STEAM across a fixed event service and also a mechanism of providing fixed event services with support for mobile clients. SIENA could be added to this mix, to extend the scale of the system geographically.

Federation allows systems to grow gracefully. So long as a requirement can be met through federation, then new features may be quickly, cheaply and transparently added to a system without modifying the existing system.

End-to-end event context integrity requirements cannot be facilitated in a general federated system because event services and the FES infrastructure (gateways, adapters) do not provide the same capabilities. However in some situations end-to-end event context integrity may be maintained. For example in Fig 7.1(a), the priority attribute of an event may be preserved across a FES system composed of a CNS event service and a JMS event service. Here this is only one input event service and one output event service in each direction. Events are delivered in priority order from one event service to the FES. The FES processes these events in the same order and forwards them to the other event service in the same order with the priority attribute mapped to the same relative priority. Such bilateral inter-working of similar event services might be a common usage of the FES (e.g. CNS/JMS). Fig. 7.1(b) is a logical extension of (a). Gateways could be extended to maintain event context integrity between event services of similar capabilities for common event attributes.

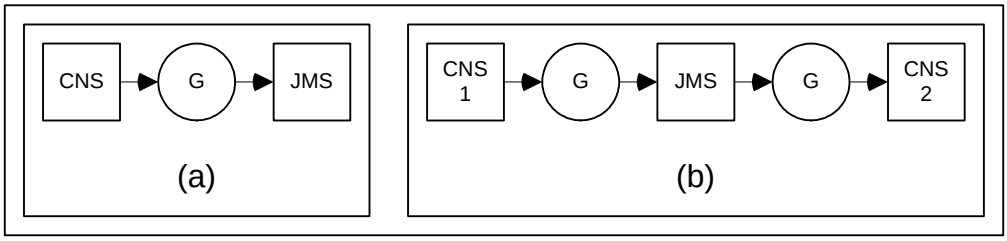

Fig 7.1 Addressing end-to-end event context issues

As described in the overview chapter, federation can meet scalability, load-balancing, fault tolerance and performance requirements. This might be the compelling reason to choose a federated solution, as these requirements may be difficult to meet using other methods, especially in very large systems.

> *"Typically, this situation arises in very large systems in which there are simply too many clients and objects for a single server to handle. In these situations, you have no choice except to distribute the processing load over a number of federated servers"* [14, pp1013].

7.3 Federation drawbacks

Heterogeneous event service federation cannot generally provide end-to-end event context support. The federation is only as strong as its weakest link. However, in some common situations end-to-end event context may be maintained as discussed in the viability section above.

Heterogeneous multilateral federation imposes a two-step translation process for event service inter-working – e.g. translating to the FES event model and translating from the FES event model. This may introduce a performance overhead that some inter-working requirements cannot afford.

As explained in section 5.7, it is difficult to provide a reliable request-response mechanism in a general federated system. Therefore subscribers can never be sure of the status of their subscription requests. Using an additional communication

mechanism is probably not feasible since event services exist to solve deficiencies in these other mechanisms.

Regarding event service transparency, the only time the FES solution needs to impinge on an existing event system is to discover subscription and un-subscriptions requests at run time (see section 5.3.5). Therefore the simple requirement that event services should expose these requests at run time would provide complete transparency to a federated solution such as the FES.

Lack of security mechanisms is a major drawback of a general federated solution. It would not be prudent to expose an event service in a federated system outside an organisation firewall.

The CNS specification [8, ppC-3] identifies unique message identification as an issue in event service federation. It is not clear what this means. Events could be qualified by using the identifier of their source event service. In addition to security, transaction support is also identified as complex issue.

> *Note that a conscious decision was made to not address the Notification Service RFP requirement related to Federated Channels. Essentially, satisfaction of that requirement encompasses the specification of interfaces that support creation and management of networks of connected notification channels. During the authors' discussion of that topic, we determined that it raises many complex issues related to transactions, security, and unique message identification.*

7.4 Comments on other inter-working approaches

7.4.1 Inter-working via extension

Inter-working via extension offers the possibility of one step translation, which will result in faster and more accurate translation. Because a new event service is being created, it is possible to provide all the features of the old event services. As there is only one event service, end-to-end event context integrity can be maintained.

The development effort for this approach is much more complex, time consuming and costly compared to the FES adapter approach, as the developer is exposed to the entire code base of the old event services and requires intimate knowledge of the participating event services and event service inter-working. Such a solution is also much more difficult to test, since black box and white box testing of the new event service is required. The resulting solution may be overkill for many requirements. It may be difficult to implement, use, understand, maintain and extend. Multi-lateral event service inter-working is difficult with this approach. This kind of development comes with a high risk factor.

7.4.2 Transport inter-working

Transport inter-working is a subset of the FES solution, providing event service transparent event mapping and transport. The FES solution shows how transport inter-working may be extended further to provide request mapping support, event context support and federation support.

7.5 Summary

The FES approach is an adequate and cost effective solution for many heterogeneous event service inter-working requirements.

Generally, heterogeneous event service federation, because of the flexibility and synergy gained through event service composition, can address certain system event requirements more easily and cheaply than bespoke solutions. However a general federated solution is only as strong as its weakest link. Therefore end-to-end event context integrity is impossible to maintain in a general federation.

It is very likely due to legacy issues, new technologies and new platforms that there will always be different kinds of event services with different features that may require inter-working. It is unlikely that a ‘one-hat fits all’ event service will ever emerge. Therefore an approach like the FES is vital to provide a model to interconnect these event services (derived from a discussion on internetworking in [13]).

Chapter 8. Conclusion

This chapter describes the achievements and contributions of this dissertation. Areas for future work are also outlined.

8.1 Achievements

The main aim of this project was to determine whether a standard mechanism for federating heterogeneous event services is both a valuable and realistic solution to the event service inter-working problem.

There is very little prior work regarding or defining event service federation and especially heterogeneous federation. Therefore the first achievement of this project was to determine a realistic, logical set of requirements for event service federation, based on general concepts of service federation. These requirements are outlined in the overview chapter.

Another achievement of this project was to discover and document many event service inter-working and federation issues. These issues are listed in the issues chapter.

The main achievement of this project was the design and implementation of a federation system called the “Federated Event Service”. The design and implementation was tested for proof of concept, by implementing a realistic traffic monitoring use case that demonstrates the inter-working and federation of three disparate event services: the CORBA Notification Service, the STEAM event service and the SIENA event service.

The final achievement of this project was an evaluation of the federation design and implementation to determine its benefits. In addition the viability of federation as an alternative mechanism for building and extending event systems was examined. The conclusion from the evaluation is that the FES approach is an adequate and cost effective solution for many heterogeneous event service inter-working requirements

and that federation is viable alternative to bespoke solutions for certain system event requirements.

8.2 Future work

Some possible directions for future work are listed here.

8.2.1 Tunnelling

If the source and destination event services are of the same type then the FES may be able to perform a tunnelling operation. Event service specific requests and events are wrapped into opaque packets by adapters and are transported by the FES between event services of the same type. Only the source and destination event service adapters know how to interpret the packets. There is no need to translate the request and events into the FES event model. Therefore much of the functionality of the event service may be preserved. Furthermore this method is more efficient as the data does not have to be translated as it crosses intermediate event services. Tunnelling could also be used as a short cut between intermediate event services of the same type.

8.2.2 End to end event context support

The FES cannot provide end-to-end event context support when intermediate event services do not support the necessary capabilities. FES gateways may also hinder end-to-end event context support. The current FES gateway implementation forwards events between event services without attempting to preserve the context of those events requests. FES gateways could be upgraded to recognise and maintain common, important event attribute contexts such as event reliability and priority.

8.2.3 Naming

Names are currently manually assigned within the FES. Ideally these identifiers could be determined via some automated mechanism. Perhaps something similar to DHCP [28] would work.

8.2.4 Configuration

Future versions of the FES could employ tools to create and maintain FES systems and configurations. A configuration protocol to route information to remote gateways would be quite valuable for large FES systems.

8.2.5 Monitoring

Diagnostic support is required to monitor a FES system. ICMP [29] provides this kind of support in IP networks. Maybe something similar could be applied to the FES through control events.

8.2.6 Routing

The FES could employ many IP like routing protocols to address its routing shortcomings. Automatic network configuration via spanning trees could be employed to prevent loops in the topology. A mobile IP like solution could be employed for addressing fixed to mobile event service inter-working issues. For example, use a mobile IP like solution to route requests through the nearest fixed gateway to a mobile client. The SIENA paper [6] discusses routing with respect to large-scale event server networks. Perhaps some of these ideas could be employed in the FES.

8.2.7 Event Flow Control

The FES could allow an administrator to define the allowable flow of events in a FES system. This is a desirable feature as it gives the administrator control over the inputs to and the outputs from event services in much the same way as a firewall is used to control Internet traffic. Administrators can also prevent event services from being swamped with requests and events. This feature is essential for bounded event services such as real time event services to ensure they don’t get swamped with events or requests.

8.2.8 Discovery Service

A discovery service could leverage the capabilities of a federated event service by facilitating automatic load balancing and fault tolerance. Such information that could be discovered and used include event service capabilities, event service load and event service usage cost.

8.2.9 Event type repository

An event type repository could be used to facilitate automatic event mapping. It could also be used to perform type checking.

8.2.10 Security

Security has not been covered by this dissertation. For a very good introduction to network security, please see [30]. For an overview of security issues as they pertain to event services, please refer to [31].

References

[1] R. Meier, V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems", in Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), Vienna, Austria, 2002, pp. 585-588.

[2] J. Kaiser, C. Brudna, "A Publisher/Subscriber Architecture Supporting Interoperability of CAN-Bus and the Internet", Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'2002), Västeras, Sweden, 2002.

[3] J. Kaiser, C. Brudna, C. Mitidieri, “A Real-Time Event Channel Model for the CAN-Bus”, Dept. of Computer Structures, University of Ulm, Germany, 2003.

[4] J. Kaiser, M. Mock, "Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)", 2nd Int'l Symposium on Object-Oriented Distributed Real-Time Computing Systems, San Malo, May 1999.

[5] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks", in Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), Vienna, Austria, 2002, pp. 639-644.

[6] A. Carzaniga, D. Rosenblum, A. Wolf, “Design of a scalable event notification service: Interface and architecture”, Technical Report CU-CS-863-98, University of Colorado, USA, August 1998.

[7] OMG, “CORBA Event Service Specification, version 1.1”, March 2001.

[8] OMG, “CORBA Notification Service Specification, version 1.0.1”, August 2002.

[9] Sun Microsystems, “Java Message Service, version 1.1”, April 12th 2002.

[10] Steve Trythall, "JMS and CORBA Notification Interworking", http://www.onjava.com/pub/a/onjava/2001/12/12/jms_not.html, December 2001.

[11] M. Aleksy, M. Schader, A. Schnell, "Design and Implementation of a

Bridge Between CORBA's Notification Service and the Java Message Service", University of Mannheim, Germany, Proceedings of the 36th Hawaii International Conference on System Sciences, 2003.

[12] E. Gamma, R Helms, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley, 1995.

[13] A. Tanenbaum, "Computer Networks", Prentice Hall, 4th Edition, 2003.

[14] M. Henning, S. Vinosky, "Advanced CORBA Programming with C++", Addison Wesley, 1999.

[15] Java Programming Language, Sun Microsystems, http://www.java.sun.com

[16] OMG, "Request For Proposal: Notification / Java Message Service Interworking", OMG Document: ab/2001-07-01, 2001.

[17] P.Russell, "Unified Enterprise Messaging - Seamlessly Bridging J2EE and CORBA Platforms" (presentation), IONA Technologies, http://www.iona.com, 2002.

[18] T.Urqhart, "Notification / Java Message Service Interworking Request for Proposal" (presentation), PrismTech, http://www.prismtechnologies.com., 2001.

[19] Sun Microsystems, Java Abstract Window Toolkit, http://www.java.sun.com

[20] PrismTech, OpenFusion Notification Service Connectivity Bridges, http://www.prismtechnologies.com.

[21] IBM, MQSeries, http://www-3.ibm.com/software/integration/mqfamily

[22] TIBCO, Rendezvous, http://www.tibco.com.

[23] TAO, http://www.cs.wustl.edu/~schmidt/TAO.html

[24] G. Booch, J Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.

[25] ISO, "ISO/IEC 14882 Standard for the C+++ Programming Language", Geneva, 1998.

[26] Microsoft Corporation, http://www.microsoft.com

[27] A. Tanenbaum, "Distributed Systems Principles and Paradigms", Prentice Hall, 2002.

[28] DHCP - Dynamic Host Configuration Protocol, Internet RFC 2131, RFC 2132.

[29] ICMP – Internet Control Message Protocol, Internet RFC 792.

[30] C. Kaufman, R.Perlman, M. Speciner, “Network Security. Private Communication in a PUBLIC World”, Second Edition, Prentice Hall, 2002.

[31] C. Wang, A. Carzaniga, D. Evans, and A.L. Wolf, "Security Issues and Requirements for Internet-scale Publish-Subscribe Systems", in Proceedings of the Thirty-Fifth Annual Hawaii International Conference on System Sciences (HICSS-35), Big Island, Hawaii, January 2002.

[32] LINUX, http://www.linux.org

[33] PrismTech, “OpenFusion in Financial Services”, “OpenFusion Notification Service Datasheet”, “EMS – NMS Integration using the OpenFusion Notification Service”, http://www.prismtechnologies.com.