Federating Heterogeneous Event Services

Conor Ryan, René Meier, and Vinny Cahill
Distributed Systems Group, Department of Computer Science, Trinity College Dublin, Ireland
cahryan@eircom.net, rene.meier@cs.tcd.ie, vinny.cahill@cs.tcd.ie

Abstract

As event-based middleware is currently being applied for application component integration in a range of application areas, a variety of event services have been proposed to address different application requirements. The emergence of ubiquitous computing systems has given rise to application integration across multiple areas and as a result, has led to systems comprising several, independently operating event services. Even though event services are based on the same communication pattern, application component integration across heterogeneous services is typically prevented by the constraints imposed by their respective event models.

This paper presents the design and implementation of the Federated Event Service (FES). The FES enables heterogeneous event services to cooperate and to operate as a single logical service. It therefore facilitates building event-based systems in which the application requirements cannot be met by a single event service.

1. Introduction

Event services provide asynchronous, decoupled, anonymous message-based communication. This facilitates scalable distributed systems composed of autonomous concurrently-executing entities. There are many event services in existence addressing wide ranging issues such as Internet scale (Siena [1]), quality of service (CORBA Notification Service (CNS) [2]), and mobility and location awareness (STEAM [3]). When integrating systems that use distinct event services it may be necessary to inter-work their event services to facilitate communication between the 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.

A federated service is a collection of autonomous concurrent services that may be linked together to provide a single logical service. This paper presents the design and implementation of the Federated Event Service (FES), a standard mechanism for federating heterogeneous event services. We believe that such a mechanism is a valuable solution for addressing the event service inter-working problem described above. We also believe that this 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.

The remainder of this paper is structured as follows: Section 2 surveys related work. Section 3 discusses event service federation and related issues. Section 4 describes the design of the FES. Section 5 and section 6 discuss a test implementation and usage of the FES. Section 7 concludes this paper by summarizing our work.

2. Related Work

Based on our survey of related work there seems to be little research into the area of federating or inter-working heterogeneous event services. Most of the interest, for commercial reasons, lies in the inter-working of the CORBA Notification Service (CNS) and the Java Message Service [4]. This inter-working requirement is a less difficult problem than generic event service inter-working as it is a bilateral inter-working requirement supporting similar event models, feature sets and event structures.

In [5] events are used to federate components of any granularity including event services. Similar to the FES, the approach in [6] uses a common event model and gateways to provide interoperability between event services. However, these approaches do not emphasize the issues involved with and the opportunities to be gained from heterogeneous event service federation. Our work is specifically concerned with addressing the issues involved with inter-working and federation of heterogeneous event services and aims to provide this mechanism as transparently as possible to existing systems.

3. Federating Event Services

A federated event service provides a single logical event service to clients; however, it consists of a number of autonomous event services. The federation mechanism is transparent to the participating event services and event services are unaware of other event services in the federation. Federation improves reliability – if an event service fails, the rest of the event services are still available. Services in a federation share the processing load and this can improve performance and scalability. Event services can still be administered individually. This facilitates easier management of a large service.

Many issues 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 while others are particular to event service inter-working. Important issues that we are aware of are briefly summarized here.

Event model heterogeneity: An event model consists of a set of rules describing a communication model that is based on events. Event models are discussed and classified in [7]. For a federated service to be valuable it must cater for a wide variety of event models. Important features including event propagation support, event type support, event filtering support, and event service specific features such as mobility and QoS must be considered.

Communication: Requests issued across multiple event services may be subject to failure if an individual event service fails. Allowances have to be made for the fact that mobile event services such as STEAM may be involved in the communication path. Adequate routing protocols must be considered

Naming: Event services use varying mechanisms to identify event types and instances, e.g. event channels (CORBA Event Service (CES) [8], CNS), subscription filters (Siena, CNS) and subject identifiers (STEAM, COSMIC [9]). The design must also consider identifier uniqueness across the federation, case sensitive names and varying maximum name lengths. Event service federation introduces the requirement for individual event service identification, to allow requests to be directed at a subset of the federation and events to be forwarded to the correct event services. Routing events and requests to all event services is not a scalable option. The event service federation may also require a unique identifier to allow event services to participate in more than one federation.

Scalability: How can the sizing limits of an individual event service or system be maintained when the event service participates in a federation? What performance overheads are introduced?

Security: Security in federated event services unveils many issues such as system wide administration, authentication and authorization issues. For a good overview of security issues as they pertain to event services, see [10].

Transparency: As previously explained it is important that the federation remains as transparent as possible to event services and systems. In addition event services may have multiple client applications and it may not be realistic or even possible to update them to provide support for federation due to cost, time or unavailability of source code.

Integrating semantically misaligned events: Events generated in different systems may contain identical contents but in different forms. As discussed in [6], such events may be propagated between systems assuming mapping information is available in the systems involved.

4. FES Architecture

As shown in Figure 1, a FES system consists of two or more event services and one or more *gateways* that bridge them. A gateway interfaces to each event service by means of an *adapter*. 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.

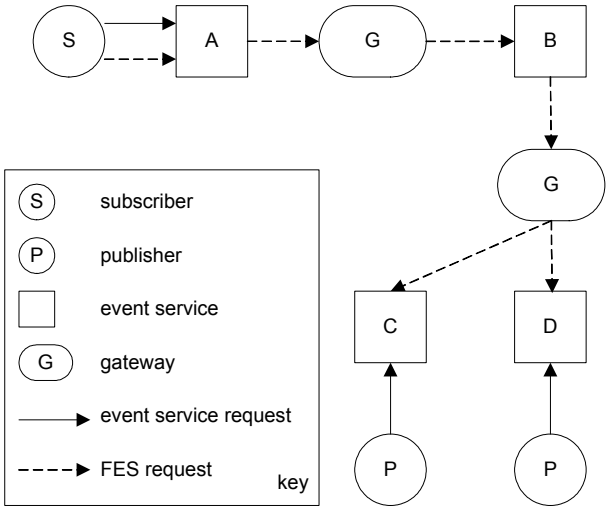

Figure 1. An example FES system.

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.* An event service that is used to route a request is known as an *intermediate* event service.

The FES supports event announcement, subscription and publication requests. These requests may be issued at any event service or gateway. The gateways propagate requests to the relevant event services. A distinction must

be made between normal event service requests as issued by a client to its direct event service (*event service request*) and requests issued by a client to event service(s) in the federation (*FES request*).

For example in figure 1, subscriber *s* issues an event service request to event service *A*. In addition *s* also issues the same subscription request as a FES request to the federation. The request is propagated to event services *B*, *C*, and *D* by the FES gateways. Any event published in the federation that matches the subscription request will be propagated back to event service *A* by the gateways and finally to subscriber *s*.

4.1. The FES Event Model

The FES event model acts as a common language between event services. To define the event model for proof of concept purposes, three basic types are required: `string`, `double` and `long`. These basic types are based on CORBA basic IDL types [11].

A FES `Event` is a structured event that is composed of a subject, a set of parameters and a set of attributes. Identifiers are case sensitive. The subject (type `string`) identifies the application event type, e.g. "DeviceOffline". The identifier (or the subject) must be unique within a FES system. Parameters contain application specific data. Attributes represent the non-functional properties of an event such as the delivery priority of an event. Parameter and attributes may be accessed by index and by identifier. 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 data and the parameter data. Attributes have the same structure as parameters. The FES does not place any limit on event size although this implementation of the FES does not support event fragmentation.

The FES supports any event service attribute that may be applied on an event-service-by-event-service basis (hop-by-hop) by adapters. The FES defines an open ended set of attributes and associated semantics, such as event delivery priority and event validity proximity. Adapters may ignore attributes that their event services do not support. Default semantics are also specified for each attribute. These defaults are applied by adapters when attributes must be supplied at an event service but are not specified in an incoming event, for example when mapping a location ignorant CNS event to a location-aware STEAM event.

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 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 [2, p.212]. Parameter/attribute access by identifier and index and case sensitive identifiers help to facilitate the mapping of event models to the FES event model.

The FES supports event filtering via the FES filtering language. At a minimum the FES filtering language must support subject based filtering. The FES approach to filtering does not depend on the extent of its filtering prowess. The FES makes use of two filters whenever a consumer makes a subscription request to an indirect event service. (1) The subscription as made by the consumer at the direct event service in the direct event service filtering language (*direct filter*). (2) 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*). The filter that is applied at the indirect event service must always define the same set of events or a superset of the events that was defined by the filter that was specified by the subscriber at the direct event service. In the case where a superset of events is specified at the indirect event service, unwanted events may cross a 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 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.

FES requests define the functions that are supported by the FES. 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.

The FES may distribute requests to one or more destination event services. Requests are one-way functions that may be applied at most once to each destination event service.

An *announcement* request 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. An *unannouncement* 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. A *subscription* request defines the events that a consumer of an event service is interested in. The consumer supplies a filter to specify this. An *unsubscription* request defines the events that a consumer of an event service is no longer interested in. The consumer supplies a filter to

specify this. 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.

A client of a FES system may specify the event services that a request is sent to via a request *distribution list*. A distribution list provides functionality that is not part of normal event services, i.e. clients may target specific sets of subscribers and publishers. 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 assuming that a suitable event service naming convention was employed. Distribution lists improve the scalability of the system.

A FES request is encapsulated in an `Event` derived `ControlEvent`. 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.

4.2. FES Gateways and Adapters

The FES is realized by a set of event services that are connected by gateways. Gateways subscribe to their direct event services via adapters for control events.

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).

The gateway must manage some local state information regarding the requests that it has made to its direct event services. For example, this includes information pertaining to subscriptions that have been made at a direct event service. When a publish request is then received from an adapter, the gateway can determine the distribution list for the event.

The adapter pattern 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. FES model mapping is an implementation detail of an adapter. Generally, there are three kinds of event mapping that an adapter may perform: user-defined (via configuration information and/or plug-in code), automatic, and combined user-defined/automatic event mapping. The integrity of a control event must be maintained at all times so requests may be applied consistently at event services. The size of control events 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.

The FES adapter interface defines five main methods corresponding to the FES requests described above. The adapter implementation must map these methods and events to event service specific functions and events. If an event service does not support announcements and/or subscriptions then null implementation can be provided for these methods. On start-up an adapter implementation must subscribe to its event service for control events. If an event service does not support filtering then the adapter must do its own filtering to single out control events. Received control events must be passed to the gateway for processing. All other events received by an adapter must be converted to publication control events before passing them to the gateway.

4.3 Using the FES

The following steps outline how the FES may be used to federate heterogeneous event services.

1) Identify the events to be propagated between event services.
2) Select and/or implement appropriate event service adapters.
3) Configure gateways with event mapping information if necessary.
4) Place gateways between appropriate event services to allow inter-event service communication to occur.
5) Event propagation between event services is initiated by forwarding a relevant subscription request to the appropriate gateway(s). This can be generally

achieved by publishing the corresponding control event to any event service in the federation.

4.4. Assessment

The FES architecture addresses some of the issues outlined in section 3. A common, flexible FES event model and the adapter pattern are used to address event model heterogeneity and naming issues. The FES is transparent to event services. Existing event service clients require modification to support dynamic FES requests. Modifications are not required to propagate a static set of event types between event services. Distribution lists can aid scalability. The FES cannot provide end to end request/event context support (e.g. QoS attributes), unless all event services in the request path provide the necessary support. Other issues are left open for future work.

5. Implementing FES Gateways and Adapters

Two approaches were considered for implementation of the FES.

In the *compiled* approach, a configuration file that describes 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. Support for different types of event services can be plugged into the tool. This approach produces 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.

The *interpreted* approach requires the development 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. This approach 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 initially implement the interpreted approach as this approach is easier to develop, test and debug. The implementation supports subject based filtering only. To test the FES design the STEAM, Siena and CNS event services were chosen as FES participants. These event services have sufficiently different event models, event services, feature sets and implementations to test the FES design. All adapters implement automatic event mapping, automatically mapping between event service structured events and the FES structured event at run time. The development platform was Visual C++ 6.0 on Windows 2000 Professional. The Win32 TAO CNS implementation was used [12, 13].

Implementing the STEAM adapter was relatively straightforward. STEAM proximity information is mapped to a FES "Proximity" attribute. The subject based filtering of STEAM easily maps to the FES filtering requirement.

The CNS adapter maps the CNS priority attribute to a FES "Priority" attribute. The CNS allows filtering on any part of a CNS structured event. The CNS adapter maps the FES event subject to the `event_name` field of a CNS structured event header.

The Siena C++ API does not support the event push 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 Siena filtering supports filtering on any parameter in the event.

6. Using FES Gateways and Adapters

The following use case describes a traffic monitoring system that is composed of three heterogeneous event services that are federated via the FES. This system monitors traffic speeds at various locations in a city and logs the license number and speed of vehicles that exceed speed limits. In this system, vehicles broadcast various events over an ad hoc wireless network using the STEAM event service that include the current speed of the vehicle. The current speed of the vehicle is published every second via a "Speed" event on the car's onboard real time network via a real time event service (RTES). This event contains the car's current speed and its license number. A FES gateway is used to inter-work the RTES 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. 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.

Figure 2 outlines the configuration of the test application that we developed to simulate this use case. Here CNS acts as the RTES. *G1* is a CNS/STEAM gateway. It is passed simulated GPS locations to 'move' it between traffic monitors. *CNS PUB* is a CNS publisher in the 'vehicle' that publishes varying "Speed" events every second to the CNS event service. *G2* and *G3* act as the roadside STEAM/Siena gateways. *SIENA SUB* is a Siena subscriber, subscribing for specific "Speed" events. The STEAM event service is collocated with the relevant gateways. On start-up a 'hardwired' subscription request is issued to G1 to specify a filter of "Speed", with a distribution list of "Cns" and with the source specified as "Siena".

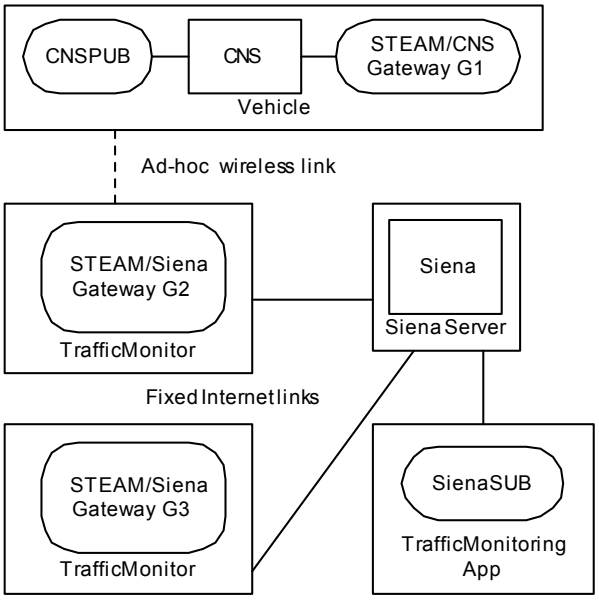

Figure 2. Test FES Application - Traffic Monitoring System.

7. Conclusions

This paper presented the design of the Federated Event Service (FES) – a system for inter-working and federating heterogeneous event services. A proof of concept implementation and test application were presented in order to show that distinct event services can be federated with the FES.

Several issues pertaining to heterogeneous event service federation were outlined and discussed. Some of the raised issues as well as issues related to request tunneling, automatic configuration, and federation monitoring remain open for future research.

References

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 283 - 331, 2001.

[2] Object Management Group, *CORBAservices: Common Object Services Specification - Notification Service Specification, Version 1.0.1*: Object Management Group, 2002.

[3] R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," in *Proceedings of the 4th IFIP Int. Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, *LNCS 2893*. Paris, France: Springer-Verlag Heidelberg, Germany, 2003, pp. 285-296.

[4] M. Aleksy, M. Schader, and A. Schnell, "Implementation of a Bridge Between CORBA's Notification Service and the Java Message Service," in *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS)*. Big Island, Hawaii, USA, 2003.

[5] J. Bates, J. Bacon, K. Moody, and M. Spiteri, "Using Events for the Scalable Federation of Heterogeneous Components," in *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*. Sintra, Portugal, 1998, pp. 58-65.

[6] J. Bacon, A. Hombrecher, C. Ma, K. Moody, and W. Yao, "Event Storage and Federation Using ODMG," in *Proceedings of the 9th International Workshop on Persistent Object Systems (POS 2000)*, vol. LNCS 2135. Lillehammer, Norway, 2000, pp. 265-281.

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

[8] Object Management Group, *CORBAservices: Common Object Services Specification - Event Service Specification*: Object Management Group, 1995.

[9] J. Kaiser, C. Brudna, C. Mitidieri, and C. Pereira, "COSMIC: A Middleware for Event-Based Interaction on CAN," in *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003)*. Lisbon, Portugal, 2003.

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

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 3.0; Chapter 3: OMG IDL Syntax and Semantics*: Object Management Group, 2002.

[12] D. C. Schmidt, "Real-Time CORBA with TAO (The ACE ORB)," www.cs.wustl.edu/~schmidt/TAO.html, 2004.

[13] T. Harrison, D. Levine, and D. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," in *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. Atlanta, Georgia, USA: ACM Press, 1997, pp. 184-200.