Notification Service - White Paper
May 2001
OpenFusion Notification Service

Introduction

Event driven applications must react to events generated by an external environment such as an arriving network packet, an updated stock quote, a news message, a telephone call, a user clicking a mouse, etc. These applications can typically not use polling to receive event notifications because this may potentially miss important event occurrences.

In particular, most distributed, object-oriented systems use event notifications since changes in one object often require dependent objects to change or react accordingly. If every object had to explicitly inform every other object about state changes the interfaces would be crowded and object implementations would be complicated, making system evolution and maintenance extremely difficult.

In order to avoid these problems, systems rely on a designated component for message dissemination. In most applications, the objects that need to communicate reside on different computer systems and are implemented in several computer languages. In addition to stringent performance and availability requirements, a messaging component must also support heterogeneous systems and open standards.

Messaging middleware that de-couples message suppliers from message consumers provides asynchronous communication between system components. Asynchronous message exchange is a prerequisite for scalability because components no longer execute in lock-step. In addition, the ability to add and remove event consumers or suppliers dynamically without affecting existing clients improves flexibility.

The first part of this paper describes the Notification Service standard submitted by the telecom task force to the OMG. The Notification Service is a very generic service for acquiring, filtering and disseminating information in distributed systems. The second part of the paper describes the OpenFusion Notification Service, which is a fast and feature rich implementation of the standard.

Background

CORBA is a standard from the OMG that defines how distributed objects can interact in a heterogeneous environment. A key component in the CORBA architecture is the ORB, which specifies how clients are able to transparently invoke operations on remote server objects. Transparent in this context means that the client is not aware of the fact that the server object potentially resides on a different host.

The interfaces of server objects are specified using IDL. There is a mapping from IDL to many popular programming languages and the implementation of a server object can be in any language supported by an ORB vendor. As object requests and replies are marshalled and sent using IIOP, the client is independent of the implementation language of the server.

Although the ORB provides a powerful means of invoking operations on distributed objects in heterogeneous environments, this functionality is insufficient for implementing sophisticated distributed applications. In addition to the ORB and IIOP specifications, the OMG has therefore defined a set of object services such as naming, event, notification and trader that cover a wide range of distributed application needs.

The Notification Service is a recent addition to the CORBA services that replaces the event service. Although the Notification Service extends the event service in a number of important
areas, it maintains backwards compatibility and builds on the same architectural principles. Before describing the Notification Service and the new features, it is therefore necessary to introduce the basic concepts of the event service.

The Event Service

The event service is a mechanism for communicating information between loosely coupled applications. Event suppliers are de-coupled from event consumers by an event channel that handles client registration and dissemination of events. Suppliers can therefore focus on application logic since the channel handles dissemination of event as well as error situations such as slow or unavailable event consumers.

The event service supports a push and a pull model. In the push model event suppliers must actively push events to the channel by invoking a suitable operation. The channel will in turn push incoming events to the connected consumers. In the pull model the channel will pull events from suppliers by invoking an operation on the suppliers. Pull consumers receive events by invoking a pull operation on the channel.

The event service requires that channels support both models as illustrated in Figure 1. Clients interact with a special proxy object that represents a communication end point. As an example, a push supplier interacts with a proxy push consumer object. This object looks exactly like a consumer (it is a push consumer by inheritance) and the supplier can thus send events as if it was connected to a single consumer.

![Event Service Diagram](image)

Figure 1: A mixture of push and pull clients connected to an event channel.

The administration objects showed in Figure 1 are factories for proxy objects. A consumer administration object can create proxy suppliers while a supplier administration object can create proxy consumers. This means that the supplier administration object represents the supplier side of an event channel, while the consumer administration object represents the consumer side of the channel.

The fact that a consumer proxy "is a" supplier and a proxy supplier "is a" consumer allows channels to be federated without using special clients that forward events from one channel to another. The inheritance structure simply allows a proxy supplier to be connected directly to a proxy consumer. A simple example of channel federation is shown in Figure 2.

Partitioning an event system into multiple "event subsystems" can have a number of advantages:

- **Performance**: Sending events to a channel that in turn forwards the events to a bulk of consumers can result in great performance improvements. As an example, if the
consumers are all on the same machine the events can be send using one network invocation and a series of local invocations.

- **Reliability**: By having multiple event channels it is possible to avoid single points of failure. Although parts of the system may no longer receive events if an event channel fails, this does not necessarily have to affect other consumers.

- **Flexibility**: Grouping of suppliers and consumers into logical units can simplify system configuration and improve flexibility. For instance, instead of changing all consumers in a group to use a new channel, only the suppliers that provide events to the group must be altered.

![Figure 2: Federating event channels can improve performance, reliability and flexibility.](image)

**Typed Event Service**

The model described so far is generic in the sense that events are sent using pre-defined push or pull operations. Furthermore, the event data is an IDL *any* variable and since the event service has no support for event filtering, all events received by the event channel will be forwarded to all connected consumers. Leveraging the dynamic invocation interface in CORBA, the event service also supports typed events.

The typed event service supports a push model and a pull model similar to the regular, un-typed event service. However, rather than using the pre-defined interfaces and operations, the clients can send or receive events using application-specific IDL interfaces. The typed event service ensures that consumers will only receive events sent by suppliers that use the same interface.

Using application-specific interfaces for message communication has a number of advantages:

- Clients can use either synchronous or asynchronous communication without re-compilation. If a system needs to be changed such that information is sent to ten instead of a single consumer, this can be done without code changes.

- Carefully developed interfaces for an application domain can be used directly when migrating an existing application to use the event service. This makes it easier to understand system operation since operation names are meaningful.

The typed event service is depicted in **Figure 3**. The clients are still required to use the proxy objects to create a connection to the event service. However, once connected the actual delivery
and reception of events is performed using the operations in the user-defined interfaces. As with the un-typed event service, channel federation can be achieved without the use of intermediators.

The typed event service uses an interface repository in order to discover the operations supported by the interface specified by clients. Since events can only be forwarded to compatible interfaces, the typed event service supports an implicit filtering. Although the use of the dynamic skeleton and dynamic invocation interfaces may decrease performance slightly, the filtering can reduce network traffic.

Figure 3: The typed event service.

Event Service Deficiencies

The event service addresses an important requirement in distributed systems, but it only provides a partial solution to the problems typically encountered. Although the fact that channels can be federated is a nice architectural feature, the event service has a number of deficiencies. Some of the more severe problems are summarized below:

- **No event filtering.** The event service does not support event filtering. This means that any event delivered by any supplier connected to a channel will be sent to all connected consumers regardless of whether they want the event or not. Since consumers may be located anywhere in a distributed system, this imposes some serious performance drawbacks in terms of network load.

- **No quality of service.** The event service has no concept of quality of service. As an example, the specification does not describe what should happen if a consumer disconnects prematurely or if a slow consumer causes an internal event buffer overflow in the event service. Other important QoS properties left as implementation details include persistent events and delivery policies.

- **No sharing of subscribed or offered types.** It is not possible for an event supplier to discover what event types connected consumers require. Also, consumers can not find out what event types are offered by the connected suppliers. Such information could be used to reduce the network bandwidth as suppliers could choose to send events only when there are interested consumers.

- **Event type safety.** In the event service, an event is an *any* variable. As any user-defined or build-in data type can be contained in an *any*, this means that there is no type checking of the data passed through the event service. The typed event service can be used for
strongly typed clients but may result in a performance penalty imposed by the dynamic invocation interface.

In addition to the above limitations, the interfaces for programming with the event service may seem a bit cumbersome in certain situations. As an example, connecting a client to an event channel involves several operations. In a similar manner it is quite involved to federate two events channels because at least eight operation invocations are required.

**The Notification Service**

The Notification Service is a recent addition to the growing number of OMG specifications. The specification addresses all of the deficiencies mentioned in the previous section while retaining backward compatibility with the original event service. This means that existing applications that use the event service can migrate to the Notification Service with no or few code changes.

The basic functionality of the Notification Service is the same as for the event service, i.e. the service is a mechanism for delivering events from a number of event suppliers to a number of event consumers. As with the event service, suppliers are de-coupled from consumers by means of an event channel and the basic components (proxy, administration object and channel) are the same in the Notification Service.

In addition to the usual *any* type, the Notification Service supports both structured and sequence type consumers and suppliers. A structured event is a well-defined IDL structure that consists of event type information, a number of filterable header fields and a message body. Sequence type clients can send or receive messages in batches to improve performance.

The Notification Service is outlined in Figure 4. The figure does not include all the different types of consumer and supplier types but illustrates the concept of multiple administration objects. As with the original event service, the Notification Service event channel supports reception and delivery of events in any mixture of the different supplier and consumer types, and communication styles.

![Figure 4: The components of the Notification Service.](image-url)

The architecture of the Notification Service imposes a certain grouping of the objects within a channel. A channel serves as a logical group for all administration objects created by the channel and all proxy objects created by those administration objects. In a similar manner an
administration object serves as a logical group for all the proxies created by the administration object in question.

The state transition diagram for proxy pull consumers and proxy push suppliers is shown in Figure 5. The ability to suspend and resume a connection makes event communication very flexible. A supplier proxy will queue up any events received, but the events will not be delivered until the consumer resumes it. In a similar manner, a consumer proxy will not attempt to pull any events from a supplier while suspended.

![State Transition Diagram](image)

Figure 5: The state transition diagram for a Notification Service proxy.

**Event Filtering**

One of the most important improvements over the original event service is support for event filtering. As an alternative to the typed event channels, the Notification Service introduces a **structured event**, which is an IDL data structure. A structured supplier or consumer is sending or receiving structured events rather than the plain any type. Figure 6 shows the contents of a structured event.

```
Fixed header

| domain_name | type_name | event_name |

Variable header

| chf_name1 | chf_value1 |
| chf_name2 | chf_value2 |
| ... |
| chf_name_n | chf_value_n |

Filterable body

| fd_name1 | fd_value1 |
| fd_name2 | fd_value2 |
| ... |
| fd_name_n | fd_value_n |

Remaining body

| remainder_of_body |
```

Figure 6: The contents of a structured event.
A structured event is divided into two parts:

1. **An event header.** The header is in turn composed of a fixed header and a variable header:

   a) **A fixed header.** Consists of event domain, event type and event name. The event domain identifies the horizontal domain, e.g. "transport", and the event type is a string that categorizes the event within the domain, e.g. "alarm". The event name is meant to be a unique identifier for the event.

   b) **A variable header.** Contains a sequence of name/value pairs. A name/value pair is a structure that contains a string variable and an any variable, i.e. the variable header contains named properties associated with an arbitrary value. It is intended to contain QoS properties for a specific event.

2. **An event body.** The body consists of a filterable body and a remaining body:

   a) **A filterable body.** This part is also a sequence of name/value pairs. It is intended to be used for filterable properties defined within an application domain. A client would normally construct filter constraints that apply to the properties in the filterable body area of a structured event.

   b) **A remaining body.** This final part of the structured event contains an any variable. As with the original event service, this part of the event can contain any data that a user wants to send along with the event.

The Notification Service defines a powerful constraint language that can be used to filter on any part of an event. The constraint language is an extension of the trader service constraint language used in the trader to retrieve service offers with shorthand expressions that operate on name/value pairs. Filters may be applied both at the consumer side and the supplier side of an event channel.

Filter objects can be attached to both proxy object and administration objects. Since the administration objects serve as a logical group for all the proxies created by the administration object in question, a filter on an administration object will apply to all proxies. This can improve performance significantly because an event will be filtered only once for a set of consumers with identical requirements.

The Notification Service finally supports special mapping filters. A mapping filter is an object that replaces one value contained in an event with another. As an example, a mapping filter could be used to replace the priority of an event when some criteria are satisfied. This type of substitution is useful because the event consumer may have a different opinion about the importance of an event than the event supplier.

### Quality of Service

Another important feature of the Notification Service is the configurable QoS properties. An implementation of the Notification Service is not required to support all the different QoS properties described in the specification, but an implementation that does will cover a very wide range of different application requirements. The standardized QoS properties can be divided into four categories:

1. **Reliability.** In the Notification Service, the reliability of an event channel is divided into two areas:

   a) **Event reliability.** Events are delivered using a best effort or guaranteed policy. Events may or may not be delivered when using the best effort policy. Guaranteed delivery implies that events are kept in a persistent store until they have been successfully delivered to all connected consumers.

   b) **Connection reliability.** Connections are transient or persistent. A transient connection will not survive Notification Service or client failures. When a connection is persistent, a supplier or consumer connection will not be lost unless the target object no longer exists.

2. **Queue management.** Three QoS properties relate to queue management:
a) **Maximum queue size.** The Notification Service supports configuration of the maximum number of events that will be stored on behalf of consumers.

b) **Delivery policy.** The delivery policy defines the order in which events are delivered to event consumers. A number of policies including FIFO, priority, and time out are supported.

c) **Discard policy.** If an event queue is exhausted, discard policies such as FIFO, LIFO, priority and time out are supported. Also, the Notification Service can be configured to discard new events.

3. **Event management.** A number of QoS properties can be set in the variable header of a structured event:

   a) **Start time.** An absolute start time can be set for each event. An event will not be delivered to any consumer connected to an event channel until the start timer expires.

   b) **Time out.** An absolute or relative time out value for an event can be specified. Events will be removed from any delivery queue when the stop timer expires.

   c) **Priority.** The priority of events can be specified. This QoS is used for queues with a priority delivery or discard policy.

4. **Batch handling.** Sequence type consumers support a few additional QoS properties:

   a) **Maximum batch size.** The maximum number of events that a sequence type consumer wishes to receive at a time. This value is independent of how suppliers deliver events to a channel.

   b) **Pacing interval.** The maximum time consumers will wait for events. If fewer events than the batch size are available after the pacing interval, the consumer receives whatever events are available.

As with filters, the QoS properties can be applied in the context of logical groups. As an example, the initial QoS properties of an event channel object will apply to all the administration objects that are subsequently created by the channel. However, fine-grained QoS is also supported, e.g. different delivery or discard policies can be specified for individual proxies.

**Subscribed and Offered Types**

The structured event type makes it possible to distinguish event types. The type of an event is composed of the event domain and the type name from the fixed event header and the Notification Service uses event type information to limit the number of events it needs to process. As an example, the Notification Service can immediately discard an event if no consumers are interested in receiving events of that type.

In addition to the filter constraints described earlier, a filter also contains a sequence of event types it can be applied to. This is necessary as constraint expressions can typically only be applied to certain event types. The Notification Service will only apply a filter to a structured event if the domain name and type name fields of the fixed header match one of the filter event types.

Suppliers and consumers can inform the Notification Service about event types either directly by invoking an operation with the supported or required types, or indirectly by means of the filter objects attached to proxy and administration objects. The aggregate of event types is called the **offered types** for suppliers and the **subscribed types** for consumers. Both offered types and subscribed types can be modified at run-time.

The event channel maintains an aggregate of all event types offered by all suppliers and all event types required by all consumers. If the aggregate changes the channel will notify all consumers or suppliers about the new offered or subscribed types by means of a callback operation. This functionality can be disabled if a client is not interested in a callback when the aggregate changes.

Intelligent suppliers and consumers can do useful things in this callback operation in order to limit the number of events sent by the Notification Service. As an example, if no consumers are interested in an event type, a supplier could decide to stop sending it. In a similar manner,
intelligent consumers could disconnect from a channel when the events of interest are no longer offered.

**Event Type Repository**

The event type repository contains meta data about event types. Because the repository was specifically designed to fulfill the requirement of verifying filter constraints, the repository only contains information about the properties in the filterable body of a structured event. Figure 7 shows the UML model for the event type repository.

![Figure 7: The UML model for the event type repository.](image)

The repository is a singleton that supports a number of event domains and contains a number of event types. An event type has a domain, a name and a number of properties. Event types can inherit or import the properties of other event types. The model in Figure 7 is mapped to IDL using the MOF, i.e. the resulting interfaces have reflective capabilities and the repository can be accessed with generic tools.

An important property of the event type repository is the ability to modify event types and the relationship between event types at run-time. This allows applications to evolve over time, e.g. new event types with additional properties can inherit from existing event types. New applications can take advantage of the additional information, while existing applications process the event according to the old set of properties.

As the event type repository is populated, it will contain the properties that are expected for the event types used in a distributed application. Clients can look up event types and investigate what properties are available for filtering. Thus, event suppliers can find out what properties are expected in the events they produce and consumers can use the repository to create meaningful constraints for event filtering.

**Management of Event Domains**

The management of event domains specification is a standard defined by the telecom task force within the OMG. It defines common interfaces for simplified connection of clients to an event
channel as well as interfaces for federating event channels. An event domain can manage any number of event channels, whether the channels are connected or not.

The motivation for developing the event domains specification was that connecting clients to a Notification Service event channel is somewhat involved using the standard interfaces. A client needs to locate a channel, get or create the administration object and finally create and connect a proxy object. The event domains service allows clients to perform these actions using a single operation.

As described earlier, the Notification Service architecture has native support for channel federation. Setting up a connection between two channels requires that the three steps described above are repeated for both channels. The administrative burden of setting up a federated event system is greatly simplified using the event domains service because two channels may be connected with a single operation.

Arbitrary connections may be set up between channels resulting in directed graphs that contain cycles and diamonds. Since there are some practical problems associated with uniquely identifying events and detecting cycles and diamonds at runtime as events propagate through a federated network of channels, the event domain service can simply be configured to reject creating connections that form these topologies.

**OpenFusion Notification Service**

The OpenFusion Notification Service supports all interfaces described in the specification, including the typed Notification Service interfaces, the event type repository and management of event domains. Also, the OpenFusion Notification Service supports all the QoS properties and thereby a very wide range of application requirements. The service is implemented in Java for platform independence.

The OpenFusion Notification Service is built on top of a generic server framework, which is depicted in Figure 8. At runtime, the server framework has the role of a container and it was designed to be flexible and configurable. It provides support for basic functionality such as object lifecycle management, logging, and system management.

Each functionality area of the server framework supports various plug-in components that may be configured by developers at deployment time. As an example, a server can be configured to use a database or flat file for storing information related to persistent CORBA objects. The plug-in architecture is extensible, i.e. application developers can create a custom plug-in to suit specific requirements.
In addition to the common plug-in framework, an OpenFusion server may support other plug-ins. The Notification Service has a plug-in for storing persistent event information as well as a plug-in for event filtering. The latter allows application developers to implement in-process custom filtering mechanisms that may give tremendous performance improvements over the generic constraint parser.

The Notification Service currently supports two plug-ins for filtering events:

1. The standard extended trader constraint language. This is a very generic grammar that supports filtering on any part of an event.
2. An SQL92 grammar. This is a simpler (and faster) grammar that supports filtering on the name/value pairs in the filterable body of structured events.

Figure 8: The OpenFusion server architecture.
The OpenFusion Notification Service is illustrated in Figure 9. Conceptually, each channel is a queue that disseminates events to a set of delivery queues. The master queue supports event persistence using a plug-in component and the delivery queues support filtering based on the filter plug-in modules. The queues support an extensive range of QoS properties to support very diverse messaging requirements.

The Notification Service specification allows vendors to support additional QoS properties as value added features without introducing incompatibilities. The QoS interfaces have operations for negotiating and obtaining the QoS settings supported by a particular implementation. The OpenFusion implementation of the Notification Service supports the following additional features:

1. **Thread pool management.** The implementation is based on a thread pool in order to allow scalability for large number of consumers. The thread pool can be configured as follows:
   a) **Thread pool size.** The maximum thread pool size.
   b) **Thread idle time.** The maximum time a thread is allowed to be idle before it is terminated.

2. **Connection management.** As mentioned above, when using persistent connections, the Notification Service will keep retrying connections until the target object no longer exists.
   a) **Reconnect interval.** This is the delay between retrying failing persistent connections. This QoS should be set to a value that corresponds to the typical time it takes for a consumer to re-start.
   b) **Maximum inactivity interval.** Proxy objects will be garbage collected when this interval of inactivity has expired.

3. **Advanced queue management.** Allows clients to interact more intimately with the Notification Service queues than is allowed by using the proxy interfaces.
   a) **Acknowledge modes.** Clients may specify automatic, manual or lazy acknowledge modes.
b) **Acknowledge level.** Message acknowledgement can be set to local or global, indicating whether a message acknowledge affects a single consumer or all consumers connected to a queue.

c) **MaxUnacknowledged.** When using manual acknowledgement, the maximum number of events that will be delivered by a delivery queue to its attached listener can be set.

4. **Dead letter drop.** The OpenFusion Notification Service allows events that have been discarded from a proxy queue to be stored for later inspection.

5. **Memory Management.** Memory usage can be controlled using two related properties.
   a) **MaxMemoryUsage.** The number of bytes at which memory usage should be controlled using the current policy.
   b) **MaxMemoryUsagePolicy.** Memory usage can be controlled when an event is delivered to an event channel either by purging event data from memory, discarding events or by raising an exception.

6. **Backwards Compatibility.** The OpenFusion Notification Service supports the very latest “Recently Adopted Specification” of the OMG Notification Service - version 1.3. This version contains two important changes to the semantics of earlier version of the specification, so two additional QoS properties are provided in order to allow backward compatibility with the old semantics.
   a) **DisconnectCallback.** This property defaults to true. If set to false then proxies will not invoke the disconnect operation of their attached client when their own disconnect operations are invoked.
   b) **AlwaysPull.** This property defaults to true. If set to false then pull suppliers will not have events pulled from them if there are not currently any attached push consumers or pull consumers attempting to retrieve events.

The OpenFusion Notification Service supports various adapters that allow interaction with other messaging systems and API's. The current version supports JMS for Java clients since this is the preferred API for accessing messaging systems from a Java platform.

The OpenFusion Notification Service finally comes with a number of graphical management tools that makes it easy to manage and supervise the service. These tools allow a client to interact with all Notification Service objects, including event channels, administration objects, filters and proxies. The OpenFusion Notification Service product also supports graphical suppliers and consumers for testing an event channel.

**OpenFusion Enterprise Message Server**

OpenFusion Enterprise Message Server is a robust and scalable messaging backbone which provides integration between back–end applications and the new front-end EJB components using an asynchronous messaging architecture. It implements key open standards and provides transparent integration between different messaging systems, including IBM’s MQSeries, TIBCO Rendezvous and third party JMS products. The OpenFusion Message Server provides its own outstanding implementation of the Java Messaging Server (JMS) and combining this with its seamless integration with the Notification Service allows messages to be sent and received to the vast world beyond CORBA.

At the heart of the OpenFusion Message Server is the Notification Service messaging engine which has been described in some detail within this paper. It is an extremely powerful and flexible CORBA based engine which allows messages to be supplied and consumed by a wide range components written for different platforms and in different languages.
Conclusion

The OMG Notification Service is a powerful mechanism for sending event notifications between applications in distributed and heterogeneous systems. The Notification Service is an extension of the original event service that provides backward compatibility with the event service and allows easy partitioning of an event system into logical groups that communicate using federated event channels.

The Notification Service overcomes a number of limitations with the event service. An important extension is the support for event filtering which can be used to reduce the number of events disseminated to consumers. Although filters can apply to any part of an event, consumers will typically filter on the filterable body of structured events, which contains a sequence of name/value pairs.

Also, the Notification Service supports a large number of configurable QoS properties. These can be divided into reliability, queue management, event management and event batch management. The Notification Service supports coarse or fine-grained QoS settings as different QoS properties can be applied all the way from the channel level down to individual events.

The Notification Service also informs event suppliers and consumers about changes to the event types required or offered by a channel. The channel obtains information about the event types implicitly via filter objects or explicitly from the connected consumers and suppliers. Intelligent suppliers and consumers can use this offer and subscription information to limit the amount of events sent across the network.

An optional part of the Notification Service specification is the event type repository. The repository helps to maintain an evolving set of event types while a complex distributed system is being constructed. More importantly, for systems that evolve during their lifetime, the event repository allows event suppliers and consumers to add additional event types or obtain information about the properties of existing types.

In addition to being a full implementation of these mandatory and optional parts of the Notification Service specification, the OpenFusion product supports a number of additional features that address enterprise requirements. The OpenFusion Notification Service supports a configurable persistence mechanism as well as QoS properties for connection time out, event acknowledgement and general queue management.

Finally, the Notification Service supports the management of event domains specification, which simplifies programming with the Notification Service. Using event domains, application programmers can connect clients to an event channel using just a single operation. Also, the federation of event channels is greatly simplified. The event domains service allows easy management of federated event systems.

In conclusion, the Notification Service is a powerful event mechanism for distributed systems that directly supports important properties such as flexibility, scalability, and extensibility. With support for different communication models, event filtering, a wide range of QoS properties and management of event domains, the Notification Service is an obvious candidate for the event notification component of distributed systems.
References


Abbreviations

API   Application Programming Interface
CORBA Common Object Request Broker Architecture
DII   Dynamic Invocation Interface
DSI   Dynamic Skeleton Interface
GUI   Graphical User Interface
IDL   Interface Definition Language
IIOP  Internet Inter-ORB Protocol
JDBC  Java Database Connectivity
JMS   Java Message Service
MOF   Meta Object Facility
OMG   Object Management Group
ORB   Object Request Broker
QoS   Quality of Service
UML   Unified Modelling Language
About PrismTech

For more information about the Notification Service or any of the other PrismTech OpenFusion CORBA services, please contact us at one of the addresses below.

**US Head Office**
PrismTech Corporation
6 Lincoln Knoll Lane
Suite 100
Burlington, MA
01803
USA
Tel: +1 781 270 1177
Fax: +1 781 238 1700
Email: info@prismtechnologies.com
Web: http://www.prismtechnologies.com

**European Head Office**
PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK
Tel: +44 191 497 9900
Fax: +44 191 497 9901
Email: info@prismtechnologies.com
Web: http://www.prismtechnologies.com

**Notices**

All information contained in this document is the property of PrismTech Limited. The information contained in this document is subject to change without notice and does not constitute a commitment nor liability on the part of PrismTech Limited. Please report any errors to PrismTech Limited.

No part of this document may be reproduced in any manner, including storage in a retrieval system, transmission via electronic means or other reproduction medium or method (electronic, mechanical, photocopying, recording or otherwise) without the prior written permission of PrismTech Limited.

© 2001 PrismTech Limited. All rights reserved.

All trademarks acknowledged.