Management of Event Domains – The MEDiator-Approach

Markus Aleksy, Lisa Köblitz, Martin Schader
University of Mannheim, Germany
{aleksy|koeblitz|mscha}@wifo3.uni-mannheim.de

Abstract

In this paper we describe our software component MEDiator. Our development is based on OMG’s Management of Event Domains specification. It allows the efficient management and simplified operation of different CORBA Notification Services running concurrently. After a brief introduction into the specifications of the Notification Service and the Management of Event Domains, we describe their architecture and discuss the most important interfaces. Following, we review the shortcomings of the current specification and delineate our approach to solve the problems that result from these deficiencies.

1. Introduction

The Common Object Request Broker Architecture (CORBA) Standard [6] defined by the Object Management Group (OMG) is widely popular in the area of distributed, object-oriented applications. In addition to independence of employed hardware architecture, operating system, and programming language, this is mainly a consequence of the specification of an interoperable system architecture which governs information exchange between implementations based upon products of different providers.

The Object Request Broker (ORB) is the fundamental component for communication in distributed CORBA applications. In order to aid application developers during their work, the OMG has, furthermore, standardized a number of system-related services, the CORBAservices. These services extend the basic functionality of the ORB and can be accessed via interfaces specified by using CORBA’s Interface Definition Language (IDL).

For example, with the Event Service [2] which realizes the Publisher/Subscriber design pattern [1], CORBA was extended by an asynchronous, decoupled communication mode. In this context, different roles, namely the Publisher and the Consumer, as well as two different message models, the Push and the Pull model were defined. Furthermore, the standard differentiates a typed from an untyped model. The core of the specification is the Event Channel which acts as a mediator between the publishers and the subscribers. By specifying the CORBA Notification Service (CNS) [4], which extends CORBA’s Event Service, many necessary extensions, for example, the possibility to filter events, were later added. This specification, however, still contained several weaknesses. Problems such as coordinating the collaboration of several CNSs running in parallel were left to the developers, although such questions are of essential importance for the scalability or fault tolerance of a distributed system. Moreover, some of the CNS’s processes, e.g., connecting several event channels, are rather complex.

In order to correct the just mentioned deficiencies, the OMG published the Management of Event Domains (MED) specification [3]. MED is a supplement to the proper service specification and makes design of the CNS processes much more convenient.

An implementation of the currently available version 1.0 of the MED specification raises a considerable number of problems since the standard contains various flaws. Nevertheless, we decided to realize the specification in our MEDiator project.

2. The Architecture of the MED Specification

When joining several event channels and linking different clients with these event channels, very rapidly a rather elaborate topology whose management becomes complex can evolve. By creating an Event Domain, even complex topologies can be administered conveniently. The purpose of an event domain is managing one or more groups of interconnected event channels. These event channels can be created through already existing implementations of the CNS, which might be running in parallel on different hosts. This would entail the considerable advantage that existing programs, which use the CNS’s event channels can be extended to employ event domains without any program modifications.

By analogy to the structure of the CNS, the MED specification defines IDL interfaces for untyped event domains, managing generic, untyped event channels. Furthermore, IDL interfaces for typed event domains
that can manage untyped as well as typed event channels are provided. Moreover, the specification contains IDL interfaces for Event Log Domains managing untyped and typed event channels and Logs that are defined in the Telecom Log Service specification [5].

2.1. The IDL Interfaces EventDomain-Factory and EventDomain

The IDL interface EventDomainFactory specifies operations for creating and managing untyped event domains.

Event domains supporting the CNS’s untyped event channels are specified with the IDL interface EventDomain. Clients, i.e., Suppliers or Consumers, which want to register with an event domain, first have to connect to an event channel of that event domain.

In order to group together event channels or to connect a client to an event channel, proxy objects are used; these are based upon IDL interfaces specified in the CNS. If an event channel’s proxy supplier is connected to the proxy consumer of another event channel, then the first event channel is called Supplier Channel, and the other is the Consumer Channel. A connection between two event channels can already be set up by solely utilizing the CNS’s IDL interfaces. This approach, however, is quite laborious and needs a number of operation invocations: with the help of a ConsumerAdmin object of the supplier channel a proxy supplier is constructed and with the help of a SupplierAdmin object of the consumer channel one obtains a proxy consumer. Subsequently, the proxy consumer and the proxy supplier have to be connected to the supplier channel and the consumer channel, respectively. In the Management of Event Domains approach, these steps are combined into one single operation.

Before two event channels can be connected, they both have to be registered with the event domain and need to have obtained a unique ID. The connection itself has a specific data structure, which contains the IDs of the event channels, the ClientType, and the NotificationStyle. The ClientType determines whether the connection is to be used for events of type Any, Structured, or Sequence, whereas the NotificationStyle indicates whether the two channels will communicate using push or pull style. An event channel that has been registered with an event domain can be associated with an arbitrary number of other event channels or clients of the event domain.

By generating connections, a topology of event channels is created. This corresponds to a directed graph, where each event channel registered with the event domain is a vertex (or node) and each connection is an edge of the graph. The graph can be of arbitrary complexity; it can contain cycles or diamond shapes, meaning that the same event may reach a vertex in the graph by more than one path. Within that graph, suppliers can send events to consumers. When a supplier uses the event channel with which it is registered to send an event, the event is not only delivered to the consumers of that event channel but also to all the event channels that are connected to it. The reason is that the proxy supplier of the supplier channel plays the supplier role for the consumer channel and, vice versa, the proxy consumer of the consumer channel plays the consumer role for the supplier channel.

Before creating an event domain, it should be considered, whether cycles or diamonds in the directed graph are admissible or not. Note that cycles may result in the unpleasant consequence that events might loop endlessly through the graph, and that in topologies containing diamonds, a consumer may receive the same event more than once. It is possible to prevent such behavior by setting the Quality of Service properties CycleDetection and DiamondDetection appropriately. If, for example, the CycleDetection value is set to ForbidCycles, then an attempt to establish a connection between two event channels that would introduce a cycle will raise a CycleCreationForbidden exception.

In an event domain, information on the event types provided by suppliers, as well the event types the consumers are interested in can be stored. Each event channel contains a local database that provides information on the event types that are being offered or subscribed. In the CNS, a mechanism is defined that enables a supplier to inform all consumers on the event types that it will be propagating in the future. Here, not only has the supplier to manage communication completely, but the event channel is, subsequently, responsible for communicating the information to each consumer. The supplier merely informs its proxy consumer whereupon the event channel informs all the consumers connected to it. If a connection of this event channel to other channels exists, the information will also be passed to all the consumers of these event channels; information concerning the event types can therefore be communicated to each of the consumers in the event domain. Analogously to this “subscription_change” mechanism, an “offer_change” mechanism is built into the CNS, which enables a consumer to inform suppliers that it is interested in certain kinds of event types.
2.2. The IDL Interfaces TypedEventDomainFactory and TypedEventDomain

The TypedEventDomainFactory interface specifies operations for creating and managing typed event domains.

The IDL interface TypedEventDomain is a subinterface of the EventDomain interface and, thus, inherits all functionality from an untyped event domain. In addition to untyped communication, a typed event domain also supports a typed communication mode.

All the operations of an untyped event domain, for example registration of a client with an untyped event channel, are extended for typed event channels and for clients needing to use typed events.

If a typed connection between two typed event channels has to be formed, both event channels must have been previously registered with the typed event domain. The typed connection itself has its own, specific data structure which consists of the IDs of the event channels, the NotificationStyle (push or pull), and the name of the interface the channels will use to interact.

2.3. The IDL Interfaces EventLogDomainFactory and EventLogDomain

The IDL interface EventLogDomainFactory specifies operations for creating and managing event log domains. An event log domain maintains one or more topologies of interconnected event channels and logs, where each event channel and event log may be capable of supporting both typed and untyped communication. Logs are objects, which implement the IDL interface NotifyLog or TypedNotifyLog, respectively.

The EventLogDomain is a subinterface of the IDL interface TypedEventDomain, which, in turn inherits from the EventDomain interface. Therefore, an event log domain is a specific typed event domain and inherits all the functionality of a typed event domain, for example, adding or removing typed or untyped event channels from a domain. In addition, an event log domain defines operations for managing typed or untyped logs which are described in the Telecom Log Service [5].

3. Critical Review of the Specification

It is the MED specification’s goal to provide the definition of a simplified management structure for different CNS event channels. The MED architecture enables developers to reuse and enhance existing implementations based on the CNS. For example, when relying solely on the CNS, six operations have to be invoked to connect a client to an event channel; the MED specification defines IDL interfaces that establish such a connection with a single operation invocation. In the same way, connections between event channels can be installed easily.

According to the CNS’s rules, Quality of Service (QoS) properties that are set on the event domain level should be on a hierarchically higher level than QoS properties set on the level of event channels. If, for example, on the event domain level, the QoS property OrderPolicy was set to the value FifoOrder then any event channel registered with the event domain must send events according to the FIFO mode. However, since this hierarchy can only be supported by new implementations of the CNS, observing the hierarchy rules is sacrificed in favor of compatibility. By doing this, the general concept of QoS properties is broken.

The default values set for the QoS properties CycleDetection and DiamondDetection allow cycles as well as diamonds. This seems to be not particularly appropriate as cycles and diamonds should better be avoided in order to prevent circling or multiple deliveries of events—a point that is repeatedly underlined in the specification. Potential problems are discussed in the specification; however, no example where the admittance of cycles within an event domain would have any advantage is mentioned.

The only conceivable reason why, for example, diamonds should be allowed is that in distributed systems, one can always argue with the failure of one of the participating hosts. If various paths exist, transmission reliability can be increased. It has to be noted, however, that the number of deliveries per event, and therefore network traffic, will grow proportionally with the number of diamonds. In order to prevent unintentional network load, the default values should disallow cycles and diamonds.

In general, the current specification does not yet appear to be a conscientious piece of work. Another example: an operation get_typed_connection, which would be the counterpart of the EventDomain’s operation get_connection, is missing in the IDL specification of interface TypedEventDomain. The obvious flaws of the specification, which we will discuss in more detail in the following section, are even more serious.
3.1. Deficiencies of the Specification

Currently, only version 1.0 of the MED specification is available. That version shows several deficiencies. On the one hand, it contains several invalid identifiers; on the other hand, some of the definitions are imprecise or inconsistent. We can distinguish the following four categories:

- **Errors concerning identifiers**
  In the MED’s IDL specification, several times invalid identifiers are used. This holds, for example, for the connect operations defined in the interface TypedEventDomain, which raise the “wrong” type of exception. Errors of this kind may be detected by thoroughly studying the different documents but the implementation of the specification is made more difficult.

- **Imprecise specification of Default Channels**
  Under the current specification, it is provided that one specific event channel in an event domain is designated to be the domain’s Default Consumer Channel. In case that a consumer is registered without specifying an event channel ID it is connected to the default consumer channel. Likewise, a Default Supplier Channel has to be identified; this channel will be connected to a supplier that registers without an event channel ID. According to the specification, the first event channel that registers with the event domain is to be used as default consumer channel and also as default supplier channel. Furthermore, operations are specified, which can be invoked to install some other event channel as default supplier channel or default consumer channel later. But, it is not clarified what should happen when the default supplier channel or the default consumer channel are removed from the event domain.

- **Inconsistencies concerning exception DiamondCreationForbidden**
  In module CosEventDomainAdmin, the exception DiamondCreationForbidden is defined such that it only contains a single diamond. In the description of operation add_connection, however, one finds the following sentence: “The exception contains as data a sequence of conflicting paths, each path being a sequence of channel member identifier.” To that purpose, the type DiamondSeq, defined in the same IDL interface, would have to be used and the exception DiamondCreationForbidden would have to be defined as: exception DiamondCreationForbidden { DiamondSeq diam; }

- **Error in the Event Log Domain architecture**
  The most serious error is probably contained in the architecture of the event log domain, which should be able to manage typed as well as untyped logs. The necessary include statements which read in DsTypedNotifyLogAdmin.idl and DsNotifyLogAdmin.idl, have the consequence that a multiple inheritance structure is created, which will be rejected as erroneous by the IDL compiler.

4. Selected Implementation Aspects

In our MEDiator implementation, each of the above mentioned IDL interfaces is implemented through a corresponding class named <Interface>Impl.java (see Figure 1). The class EventDomainManagement serves as the basis for MEDiator. Depending on the command line parameters, it generates an event domain factory (EventDomainFactoryImpl), a typed event domain factory (TypedEventDomainFactoryImpl), or an event log domain factory (EventLogDomainFactoryImpl), which is able to create any number of event domains (EventDomainImpl), typed event

![Figure 1: UML Diagram of MEDiator](image)
4.1. Management of MEDiator’s Internal Data Structures

An event domain contains a group of event channels, which may be interconnected. The topology of that group corresponds to a directed graph with the event channels as vertices and the connections between channels as edges. There are several ways to represent a directed graph in a computer system, e.g., using its adjacency matrix or storing its adjacency list. The adjacency list has the advantage that its storage requirement is relatively modest: it is of order \( O(|V|+|E|) \), where \( |V| \) is the number of vertices and \( |E| \) is the number of edges. An adjacency matrix requires \( O(|V|^2) \) but simplifies the algorithms from graph theory that we employ especially for cycle detection (see Section 4.2).

4.2. Cycles

Before two event channels are connected via an invocation of operation `add_connection`, we must find out whether adding this connection will create a cycle. If cycles are not admitted, a `CycleCreationForbidden` exception must be raised.

A directed graph is called strongly connected when any vertex can be reached by any other vertex by traversing edges in the direction in which they point. An event domain’s graph is not necessarily strongly connected but it can be decomposed in strongly connected components (subgraphs). Each of these components corresponds to a group of event channels containing a cycle or it consists of a single event channel. If cycles are not admitted, before adding a connection between two event channels, we test whether a strongly connected component would be created. In that case, we would raise the exception.

4.3. Subscription and Offer Channels

The passing of messages between event channels is determined by Subscription Channels and Offer Channels. Operation `get_subscription_channels` takes as input an event channel ID and returns a list with the IDs of all event channels that can be reached from this channel. An event channel can be reached if a path to it exists in the directed graph of the event domain. If a path exists, i.e., the event channel is reachable from the input channel, then the reachable event channel is called subscription channel of the event channel passed. In order to determine the members of the list of subscription channels, a depth-first search has to be carried through. If an event channel \( A \) is a subscription channel of an event channel \( B \) then, reciprocally, event channel \( B \) is called offer channel of the event channel \( A \). The operation `get_offer_channels` is the analog of operation `get_subscription_channels`. It has an event channel as a parameter and returns the list of all offer channels of that channel; again, a depth-first search is carried through to determine the list members.

5. Conclusion

The aim to implement the MED specification in such a way that the third and highest level of standard conformity is realized was reached as planned: all modules of the specification were implemented. The main problem was posed through the considerable number of deficiencies in the specification that we discussed in Section 3.1.

Still, event domains offer a convenient extension of the CNS. Numerous operations facilitate the management of potentially very complex topologies of event channels and enable users to grasp a comprehensive overview of the topology in short time. With the help of QoS properties, unwanted cycles or multiple paths between channels can be prohibited in the first place, if requested. Connecting event channels and registering clients with event channels is rather complex if the CNS operations are employed. By introducing an event domain, this can be accomplished through a single operation invocation. Even if the specification comprises several deficiencies at the moment, the benefit of event domains is evident and convincing. Therefore, it is somewhat surprising that within three years time hardly any vendor has provided this convenient CNS extension.

References