1.0 Introduction

The Event Manager provides for a mechanism of decoupled communication in Gaia. It exports an interface for creating, deleting, retrieving event channels and sending events on a particular channel. It uses the underlying implementation of Orbacus Event Service (and the factory) for ensuring the reliable delivery of events.

2.0 Entities in the Event System

2.1 Event Manager (EM)

The EM is a Corba Wrapper over the Orbacus Event Service factory and it keeps track of what channels are currently present and what are their type and name. So it is possible to query the event manager for creating / retrieving channels.

2.2 Event Service Broker (EB)

The EB is a helper class to make the process of creating / retrieving channels easier for a developer. When it is initialized it resolves the event manager from the name service and maintains a reference to it. So all further queries should be directed to the event service broker, which in turn will route it to the event manager. It also provides an easy interface for sending events and registering for a particular channel. Typically, any client
should use the event service broker for all event related stuff. However, if it directly wishes to contact the event manager then it has to resolve the event manager in its own code. To know how this can be done please refer to the init method of EB in the UOB workspace and CorbaUtilities project.

### 2.0 Interfaces

The following paragraphs detail the interfaces provided by the event service broker.

#### 2.1 Initialization

Before using any of the methods of the EB, it is necessary to call the following initialization method on the instance of EB.

```c
int init(BaseComponent *bc);
```

In this method, a reference to the remote instance of event manager is created. It returns 0 on successful completion and a negative number on failure. Two possible causes of failure could be that the initialization of name service broker failed or the Event Manager could not be resolved.

#### 2.2 Type Creation/Deletion

```c
void createType(const char *type)
```

This method is used for creating an event channel type. It throws TypeAlreadyExists if that type already exists. The type of the channel has to be created before creating the channel.

```c
void deleteType(const char *type)
```

This method is the counterpart of the previous method and is used to delete a particular type. It throws TypeNotFound if that type does not exist.

#### 2.3 Consumer Registration

```c
CosEventChannelAdmin::ProxyPushSupplier_ptr
ESBroker::registerConsumer(const Gaia::Events::ChannelId &ch,
    CosEventComm::PushConsumer_ptr consumer)
```

```c
CosEventChannelAdmin::ProxyPushSupplier_ptr
ESBroker::registerConsumer(const char *type, const char *name,
    CosEventComm::PushConsumer_ptr consumer)
```

```c
CosEventChannelAdmin::ProxyPushSupplier_ptr
ESBroker::registerConsumer( CosEventChannelAdmin::EventChannel_ptr
    channel, CosEventComm::PushConsumer_ptr pconsumer)
```
These three overloaded methods are used to register a consumer to a particular event channel. On failure a NULL value is returned. Possible causes of failure could be that the channel to which the consumer wishes to register does not exist and in that case a ChannelNotFound exception is thrown. An AlreadyConnectedException should also be caught. On successful completion it returns a pointer to a copy of the supplier.

2.4 Event Sending

CosEventChannelAdmin::ProxyPushConsumer_ptr ESBroker::getConsumer(
    const char *type, const char *name)

CosEventChannelAdmin::ProxyPushConsumer_ptr ESBroker::getConsumer(
    CosEventChannelAdmin::EventChannel_ptr channel)

These overloaded methods are used to retrieve a reference to the consumers to which a producer sends events. If the channel identified by type and name is not found, then ChannelNotFound exception is thrown. In all other possible cases of error a NULL value is returned. The value returned from here is passed as parameter to the send method.

int ESBroker::send(CosEventChannelAdmin::ProxyPushConsumer_ptr channel,
    const char *type, const char *name, const char *EventId, const CORBA::Any &data)

This method is used to send a message on the event channel identified by channel. The value of channel is obtained by making a call to getConsumer. The structure of the event (EventInfo) sent is given below.

```
struct ChannelId {
    string name;
    string type;
};
struct Ddate {
    TTime time;
    short d; //day
    short m; //month
    short y; //year
};
struct EventInfo {
    ChannelId channel;
    string EventId;
    DDate date;
    any info;
};
```

The send method fills in the values of date and time fields and sets the payload to EventInfo.info. The ChannelId is initialized with the type and name passed as parameters to the method.

2.5 Channel Retrieval

CosEventChannelAdmin::EventChannel_ptr ESBroker::getChannel(const char *type, const char *name)

CosEventChannelAdmin::EventChannel_ptr ESBroker::getChannel(
const Gaia::Events::ChannelId &ch)

These overloaded methods are used to retrieve a reference to the event channel stored with the event manager. Combination of type and name is a unique identifier for an event channel. If the channel does not exist then ChannelNotFound exception is thrown.

Gaia::Events::ListOfChannels * ESBroker::eventChannelList()

Gaia::Events::ListOfChannels * ESBroker::typedEventChannelList(const char * type)

These two overloaded methods are used to retrieve the entire list of channels currently active with the event manager. When we pass the event type, then only channels of that particular type are returned.

3.0 Sample code

The following sample code illustrates the process of creating, deleting and retrieving event types and channels. It also shows how to use the event service broker for registering for events as a consumer and how to send events as a producer.

3.1 Consumer

/* variables */
char * type = "TestType";
char * name = "TestName";

/* initializing the event service broker . Necessary before any method on the broker can be called */
if (esbroker.init(this) < 0) {
    OS::printToConsole("%s could not initialize esbroker", MSG_PREFIX);
    return -1;
}

// creating a channel type
try{
    esbroker.createType(type);
}
catch (Gaia::Events::EventManager::TypeAlreadyExists){
    OS::printToConsole("%s The type already exists", MSG_PREFIX);
}

/* creating a channel */
try{
    esbroker.createChannel(type, "TestName");
}
catch(Gaia::Events::EventManager::TypeNotFound){
    OS::printToConsole("%s the type %s not yet created ",MSG_PREFIX, type);
```c++

/* registering the consumer for receiving events */
try {
    if (esbroker.registerConsumer(type, name, _this()) == NULL) {
        OS::printToConsole("%s could not register service consumer", MSG_PREFIX);
        return;
    }
}
catch (Gaia::Events::EventManager::ChannelNotFound) {
    OS::printToConsole("%s Exception in registerConsumer", MSG_PREFIX);
}
catch (CosEventChannelAdmin::AlreadyConnected) {
    OS::printToConsole("%s Already connected Exception.", MSG_PREFIX);
    // return NULL;
}

/* retrieving a channel */
try {
    channel = esbroker.getChannel(type, name);
}
catch (Gaia::Events::EventManager::ChannelNotFound) {
    OS::printToConsole("%s channel %s %s not found ", MSG_PREFIX, type, name);
}
catch (...) {
    OS::printToConsole("%s Exception", MSG_PREFIX);
}

3.1.1 Push Method

This method is called when an event is received at the consumer. The data is present in data.

void Consumer::push(const CORBA::Any& data) throw (CosEventComm::Disconnected, CORBA::SystemException) {
    Gaia::Events::EventInfo * event;
    Gaia::Events::TestEventStructure::TestMessage * message;
    data >> = event;
    event->info >> = message;
    // Outputing the fields of event received
    OS::printToConsole("\t\t%s Type : %s ", MSG_PREFIX, message->name);
}

3.2 Producer
```
/* variables */
CosEventChannelAdmin::ProxyPushConsumer var testEventChannel;
CosEventChannelAdmin::EventChannel_ptr ec;

try{
    testEventChannel = esbroker.getConsumer("TestType",
               "TestName");
} catch(Gaias::Events::EventManager::ChannelNotFound){
    OS::printToConsole("%s Consumer Not found ", MSG_PREFIX);
}

if (CORBA::is_nil(testEventChannel.in())) {
    OS::printToConsole("%s could not get test consumer", 
                         MSG_PREFIX);
    return -1;
}

/* sending of events */
CORBA::Any data;
Gaia::Events::TestEventStructure::TestMessage message;

message.name = CORBA::string_dup( "TestType" );
message.type = CORBA::string_dup( "TestName" );

data <<= message;
if ( esbroker.send( testEventChannel.in(),"TestType","TestName", 
                   "Exited", data ) < 0 )
    OS::printToConsole("%s Disconnected exception ", 
                        MSG_PREFIX);

5.0 Conclusion

I have tried to give a brief description of the various methods that the event service broker/event manager exports. For exact implementation details please refer to the header file ESBroker.h in the CorbaUtilities project in the UOB workspace. In the EventManager workspace there are two test projects Test and TestProducer, which give sample implementations of a consumer and producer respectively.