

### Abstract

An initial design of the basic CORBA event interface to *S* and *R* is outlined.

Asynchronous events can make programming certain applications significantly simpler. Rather than scheduling occasions in the application to determine if certain events have occurred, we wait to be notified and respond to such occasions. This potentially utilizes less resources because we simply respond on demand rather than actively probe when nothing has actually happened. In addition to the simplicity of programming and minimization of resource utilization, events are of more importance in distributed components than simpler self-contained (event multi-threaded) applications due to the quality of service issues that can arise and the potential for deadlock. See chapter 20 of [?] for a discussion of the drawbacks to the listener model and the benefits of event channels.

CORBA provides a specification for supporting event distribution in an anonymous manner that puts little burden on the event suppliers. There are a variety of models supported in the CORBA specification to suit different needs. As with much of the flexibility provided in the CORBA specification, the choices can make the entire area confusing. We will try to simplify it using the interpreted language(s) and some examples.

Consider two examples when we may use events. The first involves distributed sub-tasks that are sent to a collection of different servers to be run in parallel. Imagine that we have *S* servers and *T* tasks and that  $T > S$ . Then, we will probably send the first *S* tasks to the different servers so that each is busy and then sit back and wait for any of the servers to finish. The dispatching of the tasks is a simple CORBA call. It needs to be one-way or deferred so that the caller does not block but can continue dispatching other tasks. Then the caller must periodically check back with each server to check if the task has been completed.

Instead of this constant querying of each server's task, we can allow the server to tell us it is finished. We can specify the master in the call so that the sub-task can include a call at the end to notify the master that it is about to finish. For the reasons outlined in [?] and some other scheduling and synchronization issues, this is not as desirable as just have an anonymous delivery mechanism which notifies the master that the given task has finished. What we do is have the slave dispatch a message (in the form of a CORBA event) to a message board and have that message board notify interested parties that a new message has been posted.

There are two ways we will probably want to use the message notification in the master application. Firstly, we can sit and do nothing in that application and simply be woken up when there is a new message. In this way, the slave is not doing any other work. In this case, we want to have the message board notify the master when the message is posted. This is the push model, so-called due to the fact that the message board pushes the event to the client.

The other model is known as the pull model and this allows the master to decide when to check for a message. It allows the caller to check if a message has been posted and to go on about its business if not, and it also supports waiting indefinitely for an event to arrive. The pull-model allows the master to be doing other things and periodically check whether a task has finished.

So this identifies two different ways that we want to use event channels in *S*: a sleep-and-notify push model and an active pull model allowing periodic checks. In each case, we need to connect to an event channel supplier, either a *PushSupplier* or *PullSupplier*.

Before we use an event channel in any way, we must establish a connection with it. How we do this depends on whether we wish to act as a consumer or supplier of events and whether we want to have events pushed to us or we will pull events. We identify the channel by name or IOR as a CORBA object. If this is missing, the top-level event channel obtained by connecting to the event service is used. The *consumer* and *push* arguments characterize the nature of the connection. Finally, if we are using this as a push event consumer, we can elect to immediately start waiting for events by specifying the value **T** for the argument *now*.

```
connectEventChannel(id = NULL, consumer=T, push=T, now=F)
```

When we are finished interacting with the event channel, we disconnect from it using the the function

```
disconnectEventChannel(obj)
```

Having described how to obtain a connection, we look at how a consumer can interact with the channel. When we use a push model, we need to have a CORBA server running that implements the *PushConsumer* in the *CosEventComm*. This defines two methods, *push* and *disconnect* 'push' consumer. This is done using the regular *.CorbaServer()* mechanism and def The first of these methods is called each time we receive an event. The argument will be converted using the usual mechanism to an user-level object.

When using the pull event model, we do not need a server always awaiting events, but instead can actively retrieve events at our own convenience. There are two ways to do this. The first is to simply get an event, waiting until one is available should there not be one currently on the channel (that we haven't retrieved previously). To do this, we simply call `getEvent()`. The argument is the identifier of the event channel consumer returned from the `connectEventChannel()`. (We can do the connection in this step also, but we wait for that.)

The second approach allows us to test to see if there is an event on the channel awaiting our query. If not, then we return immediately. If there is an outstanding event, its value is returned.

```
getEvent(channel=NULL, block=T)
```

Note that using events rather than callbacks by the slave to the master avoids the need to have a method in the master's IDL. Instead, the event can be an untyped *any* object which is the return value of the task.

So how do we use the event facilities to implement the distributed task manager? First, we arrange that each sub-task return its value by pushing an event onto the common event channel shared by the manager and the sub-tasks. This is done in *S* using the function `pushEvent()`. The arguments to this function are the event channel identifier (the proxy push consumer) and a user-level object which is the value of the event. The caller can create a CORBA server object ahead of time and pass this as the value. Alternatively, a third argument can be used to specify the IDL type the object. At present, this is resolved by searching for it in the interface repository. (This may be relaxed in the future.) Given this type, the standard conversion mechanism is employed. Accordingly, there is no necessity to create the server ahead of time.

So, the body of a sub-task looks something like

```
..    # lengthy computations
..

eventChannel <- connectEventChannel(consumer=F)
pushEvent(value, obj=eventChannel)
```

or, for example,

```
pushEvent(value, "IDL:LME/Fit:1.0", eventChannel)
```

where we specify the interface for the return type.

The master task manager is a consumer of the events. Having dispatched the initial calls using the one-way call mechanism, it then connects itself to the event channel as a pull-consumer. (These are the defaults, so a simple call with no arguments could be used below.) Then, it enters a loop, retrieving events.

```
dispatchInitialTasks() # first S tasks dispatched.
eventChannel = connectEventChannel(consumer=T, push=F)

finishedTask <- 0
resultValues <- vector("list",TotalNumTasks)
while(finishedTasks < TotalNumTasks) {
  result <- getEvent(eventChannel)
  finishedTasks <- finishedTasks + 1
  # store the result into the appropriate element of
  # of the results
  idleServer <- result$server # CORBA attributed named server
  # given the server, we can find out which task
  # it was given given local information and then store
  # the result in the appropriate position.
  resultValues[\\Tt{\\}idleServer}

  if(length(remainingTasks)) {
    # pass the server to the function
    # that identifies the next suitable task.
```

```

    # This returns the reduced set of tasks,
    # with the recently dispatched task
    # removed.
    remainingTasks <- dispatchTask(idleServer)
  }
}

```

## 1 Multiple Event Channels

One detail missing from the description above is the mechanism for creating new event channels. The event service specification does not provide details of how to create multiple event channels. The *ORBacus* event service does not support this simply. Its notification service is a more flexible event handling service and does support the programmatic creation of new channels.

To overcome this slight difficulty, we can manually instantiate different event services by specifying a different port with a `--OApport`. At this point, we can use the Orbacus-specific notation

```
iiop://machine:port/object-name
```

Similarly, we can have the `eventserv` print its IOR to standard out and store this.

The **OBEvent.so** library require multi-threaded support from the `JThreads` from Orbacus. This requires compiling the `Utils`, `InteractiveR` and `S` libraries correctly. The compilation flags and the libraries specified to the linker must be carefully considered. This is true for all the libraries. One sign of a mis-configuration is that the initialization of the ORB will hang in attempting to acquire a recursive mutex.

```

3 <GPL License 3>≡
  Copyright (c) 1998, 1999 The Omega Project for Statistical Computing.
  All rights reserved.

```

```
<GPL License 3>
```