ACS 2.1 Notification Channel Implementation: Additions and Enhancements

Allen Farris
NRAO
June 26, 2003

1. Introduction

This document describes additions to the implementation of the notification channel concepts in the recent ACS release 2.1. This software is an outgrowth of previous discussions between Gianluca Chiozzi, David Fugate, and myself in late April. All the software described here is implemented in Java and uses ACS 2.1. No modifications were made to the notification channel implementation in ACS 2.1. All the software has been implemented and tested using ACS 2.1 under Linux.

2. Motivation

There were several reasons for considering these enhancements.

First, the ACS implementation of the notification channel concepts does not enforce any naming standards for applications. Naming issues are an important part of the use of notification channels because they necessarily cut across subsystem boundaries and force subsystems to agree on common structures and definitions. These additions do address these naming issues.

The ACS implementation forces an extensive use of inheritance, resulting in creating classes for the sole use of accommodating notification channel concepts. In addition, the language chosen in naming the classes reflects the use of the language in CORBA's notification channel service and not the language that is already an established part of the ALMA software ICDs. So, a second motivation was to reduce the use of inheritance and use the language that is used in the ICDs.

Another motivating factor was my belief that the ACS implementation exposes too much of the CORBA concepts and that this exposure could be reduced. To be sure, the ACS implementation does do a good job of encapsulating the use of the CORBA notification channel services, but I believe this aspect could be improved. These additions also attempt to simplify the use of the notification channel concepts and even further conceal the dependence on CORBA concepts.

In addition, these enhancements implement a "LocalNotificationChannel" class that does not use CORBA. This class can be used in the same Java virtual machine within multiple

threads that wish to communicate by publishing and receiving events. Such a class is useful in building stand-alone tools that do not need distributed computing concepts. The scheduling simulation tool will use this class and, I think, similar tools, such as the observing preparation tool, could find it useful. In fact, I would argue that such an approach should become part of ACS "lite". CORBA requires considerable computer resources, not only in memory and execution efficiency but also in added complexity, and eliminating it, for those applications that do not require distributing computing, is a definite advantage.

3. Design

Figure one depicts the design of this extension package. The function of each class and interface is described, in general terms, below.

- **Receiver** – The Receiver interface allows one to attach and detach objects to a notification channel that receive events published on that channel.
- **Publisher** – The Publisher interface allows one to publish events to a notification channel that already exists.
- **NotificationChannel** – The NotificationChannel interface is merely a combination of both the Receiver and Publisher.
- **AbstractNotificationChannel** – The AbstractNotificationChannel class forms the base class from which Local and CORBA Notification Channel classes are extended. It implements the NotificationChannel interface.
- **CorbaNotificationChannel** – The CorbaNotificationChannel class implements the notification channel concepts using a CORBA-based approach that employs the CORBA notification services.
- **CorbaPublisher** – The CorbaPublisher class implements those methods needed to craft a publisher that publishes events to a CORBA notification channel. It is an extension of the ACS 2.1 Supplier class and uses CORBA structured events.
- **CorbaReceiver** – The CorbaReceiver class implements those methods needed to craft an object that receives and processes events from a CORBA notification channel. It is an extension of the ACS 2.1 Supplier class and is intended for use within a CorbaNotificationChannel, in conjunction with the attach and detach methods.
- **EventReceiver** – The EventReceiver object is a helper class used internally in implementations of CORBA and Local Receivers. It is merely a pair -- an event type name and the receiver object used to process that event.
- **LocalNotificationChannel** – The LocalNotificationChannel class implements the notification channel concepts for the case in which multiple threads within the same Java virtual machine wish to publish and receive events.
- **LocalReceiver** – The LocalReceiver class is an internal class used by the LocalNotificationChannel. Only its Receiver methods are public. Such an object is created by static methods in the LocalNotificationChannel class.

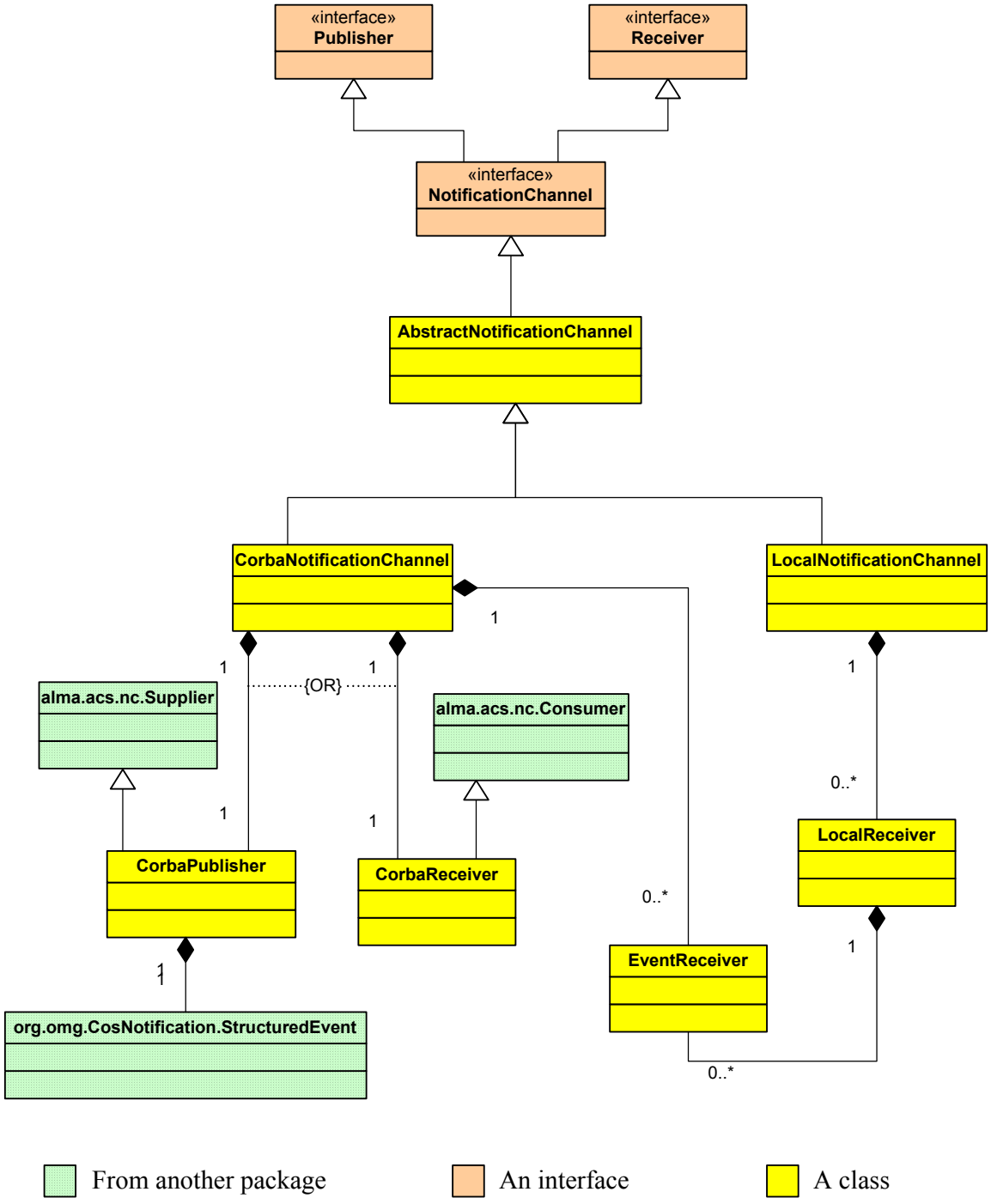

Figure 1 – Additions to the Notification Channel Implementation

4. Naming

These extensions use the language that is a part of the ALMA software ICDs. All of these have sections labeled "Published Events" and "Received Events". The events themselves are described using IDL structures. Therefore, the key concepts are Channel, Publisher, Receiver, and Event. An application creates a channel, publishes events on that channel without regard to who might be receiving those events, and other applications can choose to receive events from any channel.

These extensions assume that the application that creates the channel "owns" the channel, determining its name and properties, as well as activating and deactivating it. In creating a notification channel, the application must supply:

- The name of the channel,
- Its subsystem name, and
- An array of strings containing the event names.

The combination of subsystem name and channel name must be unique across the ALMA system. The manner in which this combination is mapped onto the CORBA notification channel name is determined by a static method in the CorbaNotificationChannel class[1]. The CORBA domain name is assumed to be "ALMA" and is hidden from the application[2].

The creator of the notification channel must supply all the names of the events that are to be published on this channel. This is not a dynamic situation; this information is known and is a part of each major ALMA software release. In the CORBA notification services such information is far more dynamic, but within ALMA, this dynamic quality is neither necessary nor desirable. The classes described here also require that the names of the events be the full-path names of the IDL structures that define the events. This is not arbitrary; this implementation employs these names and Java reflection in the process of publishing and receiving events.

A code fragment that creates a notification channel is shown below.

```
String[] eventType = new String [1];
eventType[0] = "alma.scheduling.NothingCanBeScheduled";
CorbaNotificationChannel sched =
   new CorbaNotificationChannel ("Scheduler","Scheduling",eventType);
```

[1] At present, this maps to <subsystemName>_<channelName> as the CORBA channel name.
[2] This assumption is consistent with the ACS 2.1 implementation.

This code fragment creates a CORBA notification channel with a name of "Scheduler" for the "Scheduling" subsystem. It publishes one event, described by the IDL structure, "alma.scheduling.NothingCanBeScheduled". This is consistent with the scheduling subsystem ICD.

5. The "Local" Concept – Notification Channels without CORBA

The concept of a local notification channel has been previously mentioned. The LocalNotificationChannel class implements all the methods required to publish and receive events. In fact, after being created, application code that publishes or receives events see no difference between the two. The restriction is that the LocalNotificationChannel usage is restricted to channels, publishers and receivers that exist within the same Java virtual machine. Of course, the LocalNotificationChannel does not use CORBA, which is the whole point.

A portion of the AbstractNotificationChannel class is shown below. There are two important static methods: "createNotificationChannel" and "getReceiver". By supplying either "LOCAL" or "CORBA" as a type, one can get either a local or a CORBA notification channel.

```
public abstract class AbstractNotificationChannel
      implements NotificationChannel {

public static final int LOCAL = 0;
public static final int CORBA = 1;

static public AbstractNotificationChannel createNotificationChannel (
      int type,
      String channelName,
      String subsystemName,
      String[] eventType)

static public Receiver getReceiver (
      int type,
      String channelName,
      String subsystemName)
 . . .
}
```

It might be useful to describe how this class will be used in the scheduling simulator tool. The MasterScheduler class has a parameter that allows it to configure itself as a stand-alone tool (the simulation mode) or as a part of the "real" ALMA system. This class creates or subscribes to the necessary channels either as LOCAL or CORBA. In this

manner, the simulation tool will not use CORBA and will remain a "pure" Java application[3].

6. Examples of Use

The additions described here are not intended to solve all notification channel problems. In particular, real-time classes may need to be far more dependent on details within the CORBA notification services. However, the concepts and classes implemented here are applicable to many, and probably most, of the non real-time portions of the ALMA software system, including major portions of the control subsystem.

The following sections describe the software in greater detail and illustrate its use. The most important methods are in the Publisher and Receiver interfaces that are implemented within the notification channel classes.

The Publisher Interface

```
public void publish(IDLEntity event);
```

The Publisher interface contains one method, the "publish" method that publishes an event on the notification channel. The event argument must be an instance of one of the IDL structures that were specified when the notification channel was created. If it is not, an "IllegalArgumentException" is thrown.

The Receiver Interface

```
public void attach (String eventTypeName, Object receiver);
public void detach (String eventTypeName, Object receiver);
public void begin();
public void end();
```

The Receiver interface has four methods. The "attach" method specifies an event type name, which must be the full-path name of the IDL structure that defines the event, and an event receiver object. This method attaches the event receiver object to this notification channel. The event receiver object is required to have a public method called "receive(EventType)", where EventType is the name of the IDL structure that defines the event. If the receiver object does not have such a method or if the event type name is not in the list of events that can be received, an "IllegalArgumentException" is thrown. The "detach" method detaches an eventType/receiver from the notification channel. Only the specified event type is detached for the specified receiver.

[3] As of this writing the intention is also to not use ACS in the simulation tool, since at present, it is heavily dependent on CORBA.

The “begin” method must be called to initiate the process of receiving events. At this point the objects that have been attached begin receiving events. This method must be called or no events will be received. The “end” method stops all events from being processed by the attached receiver objects. All objects that have been receiving events are removed and no further events are received.

After the “begin” method has been called, the “attach” and “detach” methods may be called at any times, allowing the application to dynamically decide which events may be received and processed. However, if the “end” method is called, receivers must be re-attached and the “begin” method called again to allow events to once again be received.

Creating a Notification Channel

```
// Create the "Progress" notification channel.
String[] eventType = new String [2];
eventType[0] = "test2.idl.ExecBlockStart";
eventType[1] = "test2.idl.ExecBlockEnd";
LocalNotificationChannel progress = new
      LocalNotificationChannel("Progress","Control",eventType);
```

In order to create a notification channel, one must specify the name of the channel, the name of the subsystem, and the names of the events that are to be published on this channel. These names must be the full-path names of the IDL structures that define the events.

Publishing an Event

```
// Publish a new subarray event.
NewSubarrayState sub = new NewSubarrayState ();
sub.subarrayId = 1;
int[] n = new int [4];
n[0] = 1; n[1] = 2; n[2] = 3; n[3] = 4;
sub.antennaId = n;
Time clock = new Time(2003,4,14,16,4,0);
sub.currentTime = clock.getTime();
status.publish(sub);
```

In order to publish an event, one merely instantiates the appropriate IDL structure, populates it with data, and calls the “publish” method.

Receiving an Event

```
private Receiver exec;

// Get the channel we are going to receive.
exec =
      LocalNotificationChannel.getLocalReceiver("Progress","Control");

// Set up the receiver object, which has methods to process the
// beginning and end of SB executions.
ReceiveEvent receiver = new ReceiveEvent("Project Manager");

// Attach the receiver to the channel.
exec.attach("test2.idl.ExecBlockStart",receiver);
exec.attach("test2.idl.ExecBlockEnd",receiver);

// Begin receiving events.
exec.begin();
```

The process of receiving an event is similar to setting up event receivers on a Java GUI. First, one must gain access to the notification channel. Then one must create an object that has the appropriate methods to receive and process the events that are desired. Then that object is attached, together with the events of interest, and the "begin" method called.

7. Other Implementation Issues

An application using these classes merely sees IDL structures published and received. In fact, for the CORBA implementation these are embedded within CORBA structured events, in the filterable data portion of those events, and passed as objects of type "any". The data name used as the CORBA event type name is the full-path name of the IDL data structure.

This implementation does not employ any CORBA concepts of event filtering. It is my view that such concepts are not needed for the kinds of applications envisioned here.

These classes also rely on the ACS notification channel implementation to provide reasonable default CORBA quality of service parameters. However, provision is made to allow an application to craft whatever quality of service parameters are desired. There is an alternative CorbaNotificationChannel constructor that allows the application to supply the CorbaPublisher that is to be used in conjunction with that channel. So, one can simply extend the supplied CorbaPublisher class and override the methods that set the quality of service parameters.

8. Availability

This set of classes has not been checked into CVS, but it has been implemented and tested. A tar file, containing this extension package and two test application suites, is available upon request. It contains a Makefile that describes how to compile the source code and set up the tests. The tests require ACS 2.1.