DATA DISTRIBUTION SERVICE AS AN ALTERNATIVE TO CORBA NOTIFY SERVICE FOR THE ALMA COMMON SOFTWARE*

J. Avarias, NRAO, Socorro, NM 87801, USA
H. Sommer, G. Chiozzi, ESO, Garching bei München, Germany

Abstract

The ALMA Common Software (ACS) provides the infrastructure for the Atacama Large Millimeter Array and other projects. ACS, based on CORBA, offers basic services and common design patterns.

One of these services is the Notification Channel. Based on the CORBA Notify Service, it allows the implementation of applications based on the publisher-subscriber pattern. This is very useful for handling asynchronous messages between components, and fosters data-centric architectures and de-coupling between different components of an application.

The Notify Service has several limitations, such as being resource intensive and not scaling well with the number of subscribers.

The Data Distribution Service (DDS) provides an alternative standard for publisher-subscriber communication for real-time systems, offering better performance and featuring decentralized message processing, scalable peer-to-peer communication, and a wide set of QoS policies.

We describe the integration of DDS into the ACS CORBA environment, replacing the Notify Service. Benefits and drawbacks are analyzed. A benchmark is presented, comparing the performance of both implementations.

ACS NOTIFICATION CHANNEL

ACS [1] provides an implementation of the publisher-subscriber paradigm called ACS Notification Channel. This implementation is built on top of the CORBA Notification Service [2], specifically the TAO Notification Service implementation. It has APIs available for the three programming languages supported in ACS: C++, Java and Python [3].

The ACS Notification Channel (NC) extends the concept of the push model notification defined by the OMG Notification Service as part of CORBA, hiding as much as possible of the CORBA complexity from the developers. The ACS NC establishes QoS policies for all the channels in order to ensure that all the messages published by Suppliers will eventually be delivered to the Consumers (subscribers) attached to the Notify Service.

The ACS NC allows to establish separate communication channels on demand, when either Supplier or Consumer first try to connect to the channel. Each channel is identified by an unique string representing the Channel name. The Channels are registered in the CORBA Naming Service and they are therefore visible to any other CORBA Object.

* This work is part of Jorge Avarias' undergraduate thesis at Universidad Técnica Fedrico Santa María, Chile. It was supported by project ALMA-CONICYT grant #31060008

Notification Channel High level API

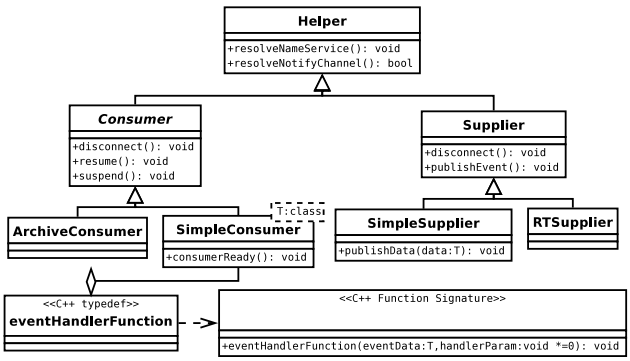

Figure 1: ACS Notification Channel API class model.

The ACS Notification Channel API, shown by the Figure 1, provides two classes to abstract the Publisher and Supplier concepts and to hide the CORBA Notification service complexity:

- `SimpleSupplier`: As the name suggests, this class is used as Supplier, it provides the method `publishData(T data)` to publish an event of the type `T` in the channel passed as argument in the constructor of this object.

- `SimpleConsumer`: This class represents the consumer, it provides the method `consumerReady()` to start to receive events asynchronously via a callback through the Notification Channel assigned in the constructor of this object.

USING DDS IN ACS FOR NOTIFICATION MESSAGING

DDS [4] is another publisher-subscriber specification that should offer better performance than CORBA Notification Service because it doesn't embeds the transmitted event data in a generic container (Notification Service embeds the data in a `Any`) and it uses a decentralized topology for events delivery providing a peer to peer communication between the subscribers allowing better scalability and better throughput than Notification Service. More over, DDS offers features like *multicast events delivery* support, new set of *QoS properties* focused in Real-Time performance

and *late joining subscriber* that allows the subscribers to receive messages sent before they had started to listen.

The new implementation of the ACS Notification Channel API based on DDS should be as transparent as possible having the same features and requirements of the previous Notify Service implementation. The requirements of the Notification Channel and a possible alternative are:

- ACS Notification Channel alternative should hide as much of DDS as possible.
- It must be possible to set QoS properties of channels.
- Event channels should never discard events and events should be delivered to consumers in a timely manner.

Also the DDS implementation should provide a similar API to the developers allowing a painless transition between Notification messaging implementations. Ideally it should be possible to use the same code and switch between the two implementations.

Classes Needed in the DDS Implementation

Using DDS, the *Supplier* concept in the ACS NC is mapped directly onto the *Publisher* class and the *Consumer* is mapped onto the *Subscriber* class. By default DDS uses the same push event notification model used in the ACS NC implementation, therefore the mapping between the concepts used in the ACS NC and the classes provided in DDS are straightforward.

The DDS notification Messaging API will expose two main classes:

- `DDSPublisher` which exposes only the `publishData` method like the `SimpleSupplier` in ACS NC API does.
- `DDSSubscriber` exposes only the `consumerReady` method like the `SimpleConsumer` class equivalent in ACS NC API.

Channel Mapping

Both publisher and subscriber must be attached to the same notification channel to be able to transfer messages. A channel name is composed by three different names that are managed by the CORBA Notification Service or the DDS Notification Message API.

- **Event Domain:** Corresponds to the "ALMA" domain. In the case of NC is represented by a string as a field in the structured event. In DDS is represented by an integer (*domain_id*) set in the middleware. In both cases it is hidden by the API.
- **Event Type:** Identifies the data type of an ALMA event. In NC, it corresponds to a field in the structured event and is determined automatically by the NC implementation. In DDS it is mapped to a topic which is created automatically when an event is sent for the first time and registered in the middleware.
- **Channel Name:** Is the entity to which the subscribers are attached. In the NC, it is represented by the CORBA Channel Object. In the DDS, it is an abstract concept that exists as a combination of Topic and Partition QoS Policy set by the subscribers.

The natural mapping for a Notification Channel would be the topic, but this approach has a problem: the ACS Notification Channel implementation must handle more than one data type for channel while a DDS topic can handle only one data type; this can be solved by mapping the channel to a partition QoS policy responsible for filtering the message in the same topic (same partition).

DDS IMPLEMENTATION

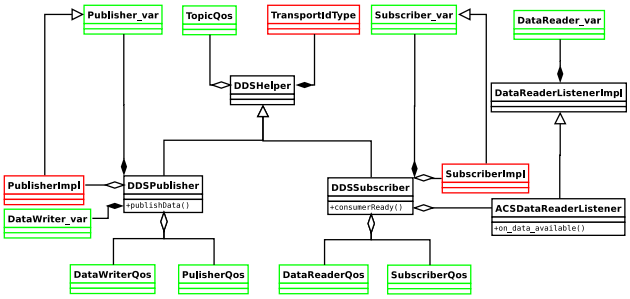

Figure 2: DDS Notification Message API class model. In the figure shows the dependencies of the API with DDS standard classes and OpenDDS specific classes.

The implementation uses OpenDDS [5], a DDS open source implementation based on ADAPTIVE Common Environment (ACE). The classes implemented in the DDS Notification Messaging API are described by the model presented in Figure 2; these classes hide the complexity of OpenDDS and offer to the developers using the ACS NC an almost identical API. The classes are briefly explained below:

- `DDSHelper`: Implements all the common procedures for the participant classes (Publishers and Subscribers). This class initializes the transport protocols, configures the topic and QoS settings and provides a proper way to dispose of the resources after having used them.
- `DDSPublisher` and `DDSSubscriber`: represent the Publisher and Subscriber implementation respectively; both inherit from `DDSHelper`.

EXPERIMENTAL RESULTS

To validate the implementation, several tests have been developed. Their main objective is to assess if DDS is a feasible and convenient option as a replacement for the current ACS implementation.

The key aspects analyzed are: *scalability*, *maximum throughput* and *resource consumption*.

To test scalability, several Subscribers and only one Publisher send events of 8 [KB] with a frequency of 10 [Hz], The round-trip time for a message is shown in Figure 3. As can be seen the DDS message trip time is more stable in comparison with NC round-trip time.

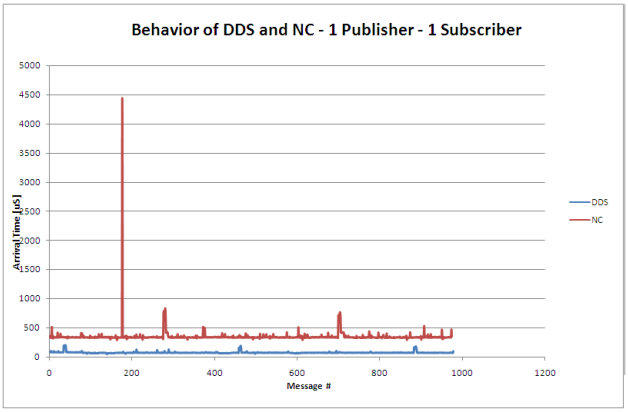

Figure 3: Comparison of both Subscribers Message Trip behavior. DDS round trip time is always below NC trip time.

A summary of the average round-trip times for a different numbers of subscribers is shown in Table 1. In both implementations, the average event trip time increases according the number of subscribers as well as the standard deviation, all this explained by the fact that the subscribers are served in sequential order. Overall the DDS implementation performs better that NC, the time taken to deliver a message is less than NC and the average deliver time increase less that in the NC case resulting in better scalability. Other DDS implementations might adopt a different dispatching strategy, using for example a pool of threads. This might drastically improve performance. We have seen for example exactly this change in the delivering strategy in the evolution of the TAO Notify Service in the past years.

Table 1: Summary Results for Scalability Test

Subscr.	**Message Trip Time Average [μs]**		**Message Trip Time Std. Deviation [μs]**	
Qty	**DDS**	**NC**	**DDS**	**NC**
1	70.32	347.63	14.44	137.96
10	304.74	1104.75	489.70	570.58
20	556.91	1925.49	297.21	1089.87
30	856.10	3154.24	470.18	1794.57
50	1596.68	5730.17	928.47	4510.51
60	2088.74	6518.49	3369.95	4337.68
75	2737.74	8143.45	1583.63	4963.38
100	3911.25	11213.95	2128.17	6278.73

In both messaging implementations the throughput test was not completed because the Publisher (Supplier) crashes before it can send all the messages requested. This behavior is expected because the clients or containers generate events as fast they can and they are not dispatched at time by the network interface, the events generated are stored in memory because by default the QoS settings are set as Reliable. In the DDS implementation the container crash can be fixed if in the Publisher sets QoS parameters to limit the message history, but if the history is limited sometimes not all messages will be received even if is the Publisher and Consumer QoS policies are set as Reliable.

The number of messages received by the clients until the container running the Publisher (Supplier) crashes can be seen in Table 2.

Table 2: Number of Messages Received and Maximum Latency in Throughput Test

	DDS	**NC**
Messages received	~ 80.000	~ 230.000
Maximum latency [μs]	$\sim 2.500.000$	$\sim 35.000.000$

Resource usage of both Notification mechanisms is shown in Table 3. The DDS main memory consumption is due to the Publisher running in an ACS C++ container, responsible for delivering the messages. With respect to the Notify Service, DDS requires four times more threads to work properly and DCPSInfoRepo requires as well a lot of threads to maintain the references between the participants.

Table 3: Memory Consumption and Used Threads for the Scalability Test with 100 Subscribers

Program	**Memory Usage [MB]**	**Number of Threads**
DDS Subscriber	4.8	12
NC Consumer	3.4	2
DDS Publisher (Container)	55.4	216
NC Supplier (Container)	3.9	7
DCPSInfoRepo	73	206
NotifyService	68.9	102

REFERENCES

[1] Gianluca Chiozzi and Matej Šekoranja, “ALMA Common Software Overview”, 2006.

[2] Object Management Group. “Notification Service Specification”. October 2004.

[3] Jim Pisano, David Fugate and Sohaila Lucero. “ACS Notification Channel (Design and Tutorial)”. ALMA 5.0.3, 2008.

[4] Object Management Group. “Data Distribution Service (DDS) for Real-time Systems”. January, 2007.

[5] Object Computing, Inc. “OpenDDS Developer’s Guide, TAO Developer’s Guide Excerpt Chapter 31”. 2007.