A CORBA event system for ALMA common software

David W. Fugate*
University of Calgary, 2500 University Dr. NW, Calgary, AB, Canada T2N 1N4

ABSTRACT

The ALMA Common Software notification channel framework provides developers with an easy to use, high-performance, event-driven system supported across multiple programming languages and operating systems. It sits on top of the CORBA notification service and hides nearly all CORBA from developers. The system is based on a push event channel model where suppliers push events onto the channel and consumers process these asynchronously. This is a many-to-many publishing model whereby multiple suppliers send events to multiple consumers on the same channel. Furthermore, these event suppliers and consumers can be coded in C++, Java, or Python on any platform supported by ACS. There are only two classes developers need to be concerned with: *SimpleSupplier* and *Consumer*. *SimpleSupplier* was designed so that ALMA events (defined as IDL structures) could be published in the simplest manner possible without exposing any CORBA to the developer. Essentially all that needs to be known is the channel's name and the IDL structure being published. The API takes care of everything else. With the *Consumer* class, the developer is responsible for providing the channel's name as well as associating event types with functions that will handle them.

Keywords: ALMA, ACS, CORBA, Notification Service, Supplier, Consumer, Event, Channel, IDL

1. INTRODUCTION

1.1. ALMA common software

Atacama Large Millimeter Array (ALMA) Common Software (ACS) is located in between the ALMA application software and other basic commercial or shared software that resides on top of the operating system(s). In a nutshell, ACS provides a generic interface between ALMA software and hardware, and it also provides basic software services common to all of the various ALMA subsystems. This consists of software developed specifically for ACS as well as operating system (OS) builds and commercial device drivers. All code developed specifically for ACS is licensed under the GNU Lesser General Public License (LGPL). Commercial and off the shelf packages are subject to their specific license agreements.

ACS is designed to offer a clear path for the implementation of applications, with the goal of obtaining implicit conformity to design standards and maintainable software. The use of ACS software is mandatory in all ALMA applications, except when the requested functionality is not provided by ACS. Motivated exceptions (for example based on reuse considerations) have to be discussed and approved on a case by case basis.

The primary users of ACS are developers of ALMA applications although ACS is rapidly gaining popularity with other astronomical software projects. The generic tools and graphical user interfaces (GUIs) provided by ACS are also used by operators and maintenance staff to perform routine maintenance operations. These tools are used to do things such as accessing logs, reading data from the ALMA configuration database, and accessing other components of the system.

Within ALMA software, subsystems have varying requirements regarding the programming languages that will be used. For example, it does not make sense for a GUI responsible for scheduling observations to be developed using the C programming language because the Control subsystem has real-time requirements. At the same time though, there must be a mechanism in place to allow subsystems to communicate with one another even though their software may be coded in different programming languages. Due to this fact, the core of ACS is based on the Object Management

* dfugate@ras.ucalgary.ca

Group's (OMG) Common Object Request Broker Architecture (CORBA). CORBA provides a clear mapping for Interface Definition Language (IDL) data types to various programming languages. It just so happens that the three primary programming languages ALMA developers use (C++, Java, and Python respectively) also have CORBA mappings.

1.2. Events and channels

The generic definition of a (software) event is an occurrence or happening of significance to a task or program, such as the completion of an asynchronous operation (www.dictionary.com). A channel is defined to be a course or pathway through which information is transmitted so we can naturally assume that an event channel is a pathway where the information consists of events (www.dictionary.com). This generic definition of an event channel does not say anything about how one sees events, makes an event occur, much less the data format the events are encoded in. There are two distinct event categories recognized within ALMA software:

1. Events passed outside a subsystem
2. Events passed within a subsystem

In the first case, there are well-defined data types to be passed between subsystems. These data types are defined as IDL structs which in turn are nearly identical to C and C++ structures. All of these IDL structs include a minimal set of related data and are intended to notify other subsystems that some software state has been reached. The conditions upon which these events occur do not really matter from ACS's standpoint. As far as ACS is concerned, all that needs to be known is the IDL struct to be published along with some sort of identification for the channel the event is to be sent on.

The other case in which events are used deals with inter-subsystem communication. For example, there might be some radio antenna data obtained from physical hardware which needs to be passed internally within the Control subsystem, but it really does not make sense to poll an object to determine when this data becomes available. In such cases, it is more useful for this information to be sent synchronously and then processed asynchronously using event channels resulting in cleaner, more efficient code. In this case also, the event data is in the form of IDL structs. This implies an identical ACS application programming interface (API) can be used for both ALMA event categories as the same basic functionality is required. From this point on, an instance of the user-defined IDL struct will be referred to as an ALMA event.

1.3. Event suppliers and consumers

Now that a working definition of an ALMA event and channel has been established, there are two more abstract terms which must be covered – suppliers and consumers. Those familiar with the observer object-oriented (OO) design pattern will recognize these two terms as subjects and observers respectively (Design Patterns). Other names that have been given to these two ideas are publishers/subscribers as well as sources/listeners. At its most basic level, an event channel supplier does exactly what its name implies: supplies events to channel(s). An event consumer on the other hand consumes these events. These two seemingly simple concepts can be far more complicated as we will soon find out.

The biggest issue dealing with suppliers and consumers is which one decides that an event needs to be published? The standard model is for an event supplier to synchronously send events which will be asynchronously consumed by some thread running within the consumer object. This is called the push channel and it begins to fall apart when consumer-side code demands new (event) data at some point in time. To illustrate this point, let's take a look at a concrete example: a data pipeline is done processing some information and needs more data from an archive. To implement this using a push channel, the data pipeline would have to send a *pipeline needs more data* event letting the archive know it is done processing the last event and needs another one. The archive in turn would send an *archive data* event containing the requested dataset only after receiving the *pipeline needs more data* event. Would not it be far simpler if the pipeline requested the *archive data* event synchronously and then blocked until the *archive data* event was received? This particular type of event channel is called a pull model where consumers synchronously pull data from asynchronous suppliers. Pull channels are conceptually the exact opposite of push channels although it is possible to mix and match the two within one channel.

1.4. CORBA events and channels

Fortunately for ACS, CORBA provides a well-defined specification for the creation, destruction, etc. of event channels. These event channels provide support for suppliers and consumers implemented in any programming language or OS platform providing a CORBA implementation. The communication between CORBA suppliers and consumers is completely decoupled, distributed, and is in the form of standard CORBA requests. Furthermore, CORBA also provides a specification for notification channels which are a superset of event channels. These notification channels add:

- Support for structured events
- Quality of service and channel administrative properties on a per-channel, per-supplier, per-consumer, and per-event basis
- The ability for published event types to be discovered by consumers at run-time
- The ability for event types requested by consumers to be discovered by suppliers at run-time
- Filtering of structured events

Within ALMA the primary reason notification channels were chosen over event channels was that notification events are much more useful in the fact that they can include more data. That is, event channel events can only be in the form of a single, simple CORBA *any* (the CORBA version of a C *void* pointer which also has object introspection capabilities), while notification channel structured events are defined like this:

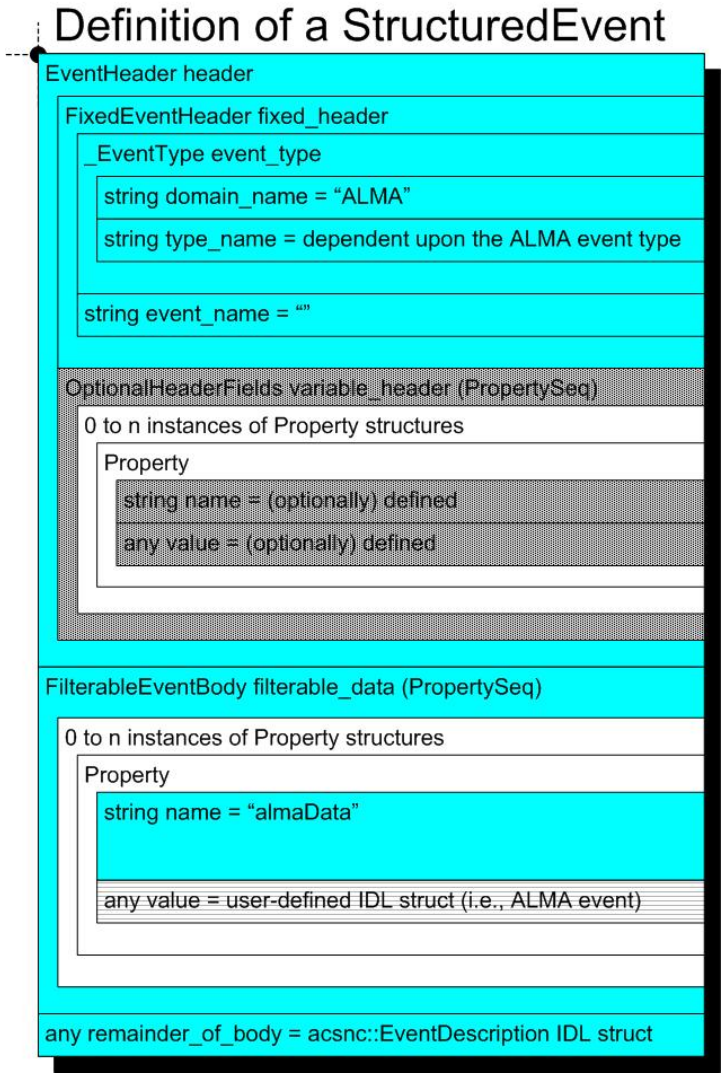

Figure 1 is a pictorial representation of a CORBA notification channel structured event with ALMA extensions added.

As you can see in figure 1, the CORBA *any* value field (denoted by a background with horizontal lines) is where the ALMA event described previously is stored within the *StructuredEvent* IDL struct. In figure 1, fields with a solid color background are automatically filled-out by the ACS APIs and fields denoted by a dotted background are not currently being used at all. For simplicity's sake, the gory details of other structured event fields will not be discussed at this time although they are covered in later sections.

A single CORBA notification channel can support any number of consumer and supplier objects connected to it (within reason). That is, if a supplier sends an event to the channel and there are multiple consumer objects subscribed to that particular event type, they will each receive a separate copy of the event. Similarly if multiple suppliers send the same type of event, a single consumer which has subscribed to that event type will receive events from all the suppliers. This is called a many-to-many event channel model and is quintessential to meeting ALMA software requirements.

Arguably the most useful feature of CORBA notification channels is the fact that events can be sent in one programming language and consumed in another. For example, a supplier object written in C++ can send events to a consumer object running within a Python interpreter. A related side note is that supplier and consumer objects are 100% decoupled from one another. They never actually communicate with each other directly and instead use proxy objects residing within the channel itself. Additionally, since the consumer and supplier objects are really CORBA objects, the supplier and consumer do not even have to reside on the same host. The same applies for the event channel which is normally run on the same host as the other CORBA services ACS utilizes.

2. ACS IMPLEMENTATION OF CORBA NOTIFICATION CHANNEL OBJECTS

While CORBA provides many ways of dealing with events, not all of them are appropriate for ALMA software. In as such, the following general standards have been adopted:

- Events are defined as user-defined IDL structs
- Channels are identified by strings
- The subsystem supplying events to a channel is responsible for the channel's lifecycle
- Push suppliers are used to supply events
- Push consumers are used consume events

The rest of this section explains in detail what the standards listed above are, why they were chosen, and what implications they may have.

2.1. ALMA events and channels

At this point we know that the high-level definition of an ALMA event is an instance of an IDL struct hidden inside a CORBA *StructuredEvent* and that ALMA events will be sent on CORBA notification channel(s). This says nothing about how event suppliers create the structured event, consumers subscribe to a particular type of event, or even how suppliers and consumers find the channel in the first place.

CORBA notification channels are nameless, remote objects that reside in the same process space as the CORBA notification service. These channels can only be located using a unique integer identification (ID) which is returned after they are created. Having supplier and consumer objects trying to agree upon an ID which is known only at run-time is not a very ideal situation. To cope with this, ALMA event channels are named using constant strings defined in IDL files. The benefits in doing this are that channel identification becomes developer friendly and by using constant strings it becomes quite difficult for suppliers and consumers to use the wrong channel name. Once the channel has been created, it is immediately registered with the CORBA naming service using the identifier provided by the ALMA developer.

When an ACS supplier object attempts to connect to a channel one of two things can occur. If the channel has already been registered with the CORBA naming service, the supplier obtains the channel reference and proceeds normally. However, if the reference cannot be found the supplier object will automatically create it. This has an

unintended side effect – the first supplier that tries to connect to the non-existent CORBA notification channel must define a set of properties which are assigned to the channel. These qualities of service and administrative properties configure exactly how the channel behaves under a variety of circumstances. A few examples are:

- *EventReliability* defines the type of guarantee that can be given for a specific event getting delivered to a consumer. Represented by *BestEffort* or *Persistent*.
- *MaxConsumers* defines the maximum number of consumers that can be connected to a particular channel at one time.
- Etc.

These properties were another driving force which led to CORBA notification channels being adopted over plain CORBA event channels. Specifically pertaining to ALMA, there are certain software subsystems that will be publishing events from real-time computers and in most cases these events will be consumed from non-deterministic machines. In other words, different subsystems will need to specify different property values for their channels. The ACS APIs support this by providing two methods (*configQoS* and *configAdminProps* respectively) in the *Supplier* class which would then be overridden in subclasses if the default property values returned are insufficient. These methods are not provided in the *Consumer* class because suppliers are currently the only objects capable of creating channels. The driving force behind this design decision was that consumer developers should not be exposed to them as the majority of properties only affect supplier-side behavior. If consumers were capable of creating channels, the major negative side effect is that consumers of events generated by other ALMA subsystems would need to ensure their *configQoS* and *configAdminProps* method implementations are synchronized with that of the supplier implementation(s).

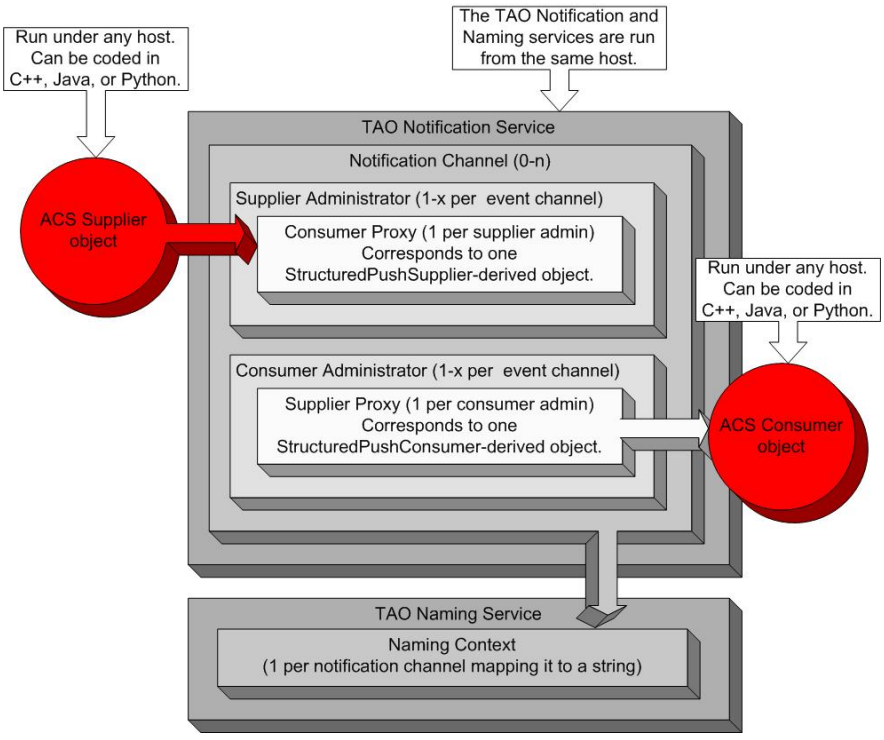

Figure 2 consists of a high-level representation of CORBA objects hidden within the ACS notification channel framework.

In section 1.4, figure 1 contains a pictorial representation of a CORBA structured event adapted specifically for ALMA software. As you can see this complex structure contains many fields which can be quite confusing and some are not even useful for ALMA purposes. Due to this fact, the ACS APIs hide all fields of this structure from developers. In the case of the *any* value field, supplier developers do supply the value but never have to explicitly set it within a *StructuredEvent* (i.e., this is hidden within the ACS infrastructure). From the value field, the API

automatically determines the *type_name* field's value. For reference, if a consumer object has not explicitly subscribed to a given *type_name*, it will never come into contact with structured events containing that specific *type_name*. The *remainder_of_body any* field will always contain an instance of the *acsnc::EventDescription* IDL struct defined by ACS. An event description is a high-level ALMA concept which quite literally provides a description of an ALMA event. In its current form, one can tell:

- Which object within a subsystem the event originated from
- When the event was published
- How many events were published by the supplier object before the current event

2.2. Supplier, SimpleSupplier, and RTSupplier classes

As previously mentioned, at the most basic level supplier objects are responsible for creating channels if they do not already exist and supplying events to channels. Please note that current ALMA requirements only require that push suppliers be implemented. Depending on the programming language a developer wants to use, at least one of three classes is available: *Supplier*, *SimpleSupplier*, and *RTSupplier* respectively.

The C++ *Supplier* class is a base class which provides a complete implementation of the *PushStructuredSupplier* IDL interface. *PushStructuredSupplier* is an interface defined by CORBA used to push structured events onto a CORBA notification channel. Only by providing a concrete implementation of *PushStructuredSupplier* can structured events be published. It provides developers with the following basic functionalities:

- Contains no direct support for ALMA events. In other words, methods are only provided to publish entire CORBA *StructuredEvent* IDL structs.
- Supplies structured events to a single notification channel synchronously. That is, the publish method blocks until the event has arrived within the notification service process. This does not necessarily mean all consumers have consumed the event after the publish method returns control.
- Can create a notification channel if it does not exist. This occurs automatically.
- Can destroy a notification channel. The associated method must be explicitly invoked to do this.
- Is notified each time a consumer object subscribes to a particular *event_type*

In Python, *Supplier* is identical to the *SimpleSupplier* class which is discussed in the next paragraph.

The *SimpleSupplier* class, provided in C++, Java, and Python, serves one purpose within the ACS notification channel framework – support publishing ALMA events while hiding 100% of CORBA code from developers. It provides the same methods as *Supplier* but also includes an additional method designed to publish an ALMA event instead of the somewhat bewildering CORBA *StructuredEvent*. From the IDL struct instance, *SimpleSupplier* automatically determines what the *type_name* field of the *StructuredEvent* should be for the developer. In summary, things could not be any simpler for the developer that chooses to use the *SimpleSupplier* class and is satisfied with default values for most things.

RTSupplier, much like *SimpleSupplier*, is derived from the *Supplier* class and provides direct support for publishing ALMA events. *RTSupplier* is designed to be used in real-time software as the prefix RT might suggest. A novel approach has been taken here: events are not published in real-time as the name of this class suggests. There are no ALMA software requirements that state an ALMA event must be received by consumer objects within a certain amount of time. Instead, when the developer has called the publish method the CORBA structured event is placed in a queue and there is no immediate network traffic. Some time later a low-priority thread wakes up and publishes as many events from the queue as it possibly can before being swapped out by the scheduler for another task. In doing this, an instance of *RTSupplier* can safely publish events from real-time code without fear of long network delays. Other than that, a great deal of effort was put into discovering processor intensive code segments within the *SimpleSupplier* class and removing them from *RTSupplier*.

2.3. Consumer and SimpleConsumer classes

ALMA event consumer objects must be able to attach themselves to channels, subscribe to events, and do *something useful* with the event data when it arrives asynchronously. Once more support of pull consumers has been omitted as no ALMA subsystems have requested this feature yet. Connecting to channels defined by a string and

subscribing to events defined by IDL structs are fairly trivial, but what exactly does *something useful* mean? As far as ACS is concerned, it does not really matter as this is an ALMA subsystem decision. However, ACS must provide a mechanism to extract the ALMA event from the CORBA structured event and deliver it to the developer in a nice way. For this purpose, the *Consumer* and *SimpleConsumer* classes described in the proceeding paragraphs were created. In short, *Consumer* was designed to be subclassed by the developer when a single consumer object is responsible for subscribing to many different types of related ALMA events on a single channel. *SimpleConsumer* is more useful when there are few ALMA event types that will be subscribed to or the data found within the events is unrelated to other types that have been subscribed to.

The *Consumer* class provides a concrete implementation of the *StructuredPushConsumer* IDL interface needed to consume CORBA structured events from a channel. The general functionality provided in *Consumer* is as follows:

- Connects to only one channel implying events sent from other channels can never be received
- Only receives the type of ALMA events that have been subscribed to
- Has the capability of unsubscribing from an ALMA event type at any time
- Can filter out entire structured events based on the data contained within them
- Has the capability of stopping and resuming receiving all events
- Is notified when a supplier starts or stops publishing an event type
- Includes the implementation of the CORBA method invoked by the channel itself (i.e., *push_structured_event*) each time an event is received. This implementation automatically extracts the ALMA event and passes it to an ACS-defined *Consumer* method (i.e., *processEvent*) which should be overridden by the developer.

Unlike the *Supplier* class, developers choosing to override *Consumer* will be exposed to no CORBA code. The Python and Java *Consumer* classes also provide the complete functionality of the *SimpleConsumer* class described in the next paragraph.

Much like the *SimpleSupplier* class, the main goal behind *SimpleConsumer* was to hide all CORBA code from ALMA developers and to do this without requiring subclassing. This has been achieved through the use of handler functions also referred to as receiver objects under the ACS Java notification channel API. The basic idea here is that when a developer creates an instance of *SimpleConsumer* and begins subscribing to various ALMA event types, they also provide a reference to a function or method which takes in an instance of the ALMA event type as its parameter. Within the *SimpleConsumer* class there is an associate array mapping *event_type*'s to their handler functions. Each time an event is received the *SimpleConsumer* instance first checks to see if the developer has mapped a function to the ALMA event type. If this is the case, *SimpleConsumer* invokes the function using the ALMA event extracted from the *StructuredEvent*. If for some reason a handler function has not been registered to handle the ALMA event or the handler function throws an exception, *SimpleConsumer* calls the *processEvent* method assuming it has been overridden by the developer.

2.4. ACS event administrator interface

Shortly before the release of ACS 3.0 it was found that some ALMA events that should have occurred were not actually being received by consumer objects. After some investigation it turned out that a few subsystems had created new IDL structs for the ALMA events and the dependent subsystems' consumers were not updated to reflect these changes. With the ACS 3.0 notification channel infrastructure, there was very little chance of detecting such anomalies. In as such, the *ACSEventAdmin* IDL interface was created and implemented in the Python programming language for ACS 3.1. *ACSEventAdmin* was designed to administer ALMA event channels and also to monitor **all** ALMA events. It supports the following basic operations:

- Creating new notification channels
- Reconfiguring the quality of service and administrative properties of existing channels
- Destroying notification channels
- Obtaining a list of all active channels

In addition to the functionality described above, there are a couple of non-trivial methods provided. The first being the *getChannelInfo* method which returns various pieces of information relevant to a single channel. Using this method one can determine the precise number of suppliers and consumers currently publishing and receiving events. Regarding events - the total number of events sent on the channel along with a list of *event_type*'s sent so far is also returned. The

second method is a great deal more useful but harder to understand conceptually. *monitorEvents* literally monitors all events from all channels using the OO callback design pattern. For those unfamiliar with callbacks, this design pattern is most useful in situations where some method takes an inordinate amount of time to finish, but the callee needs control returned immediately after invoking the method. The idea here is that a reference to a callback object existing in the callee process space is provided as an additional parameter to the method which then returns control immediately after being invoked, thereby making it asynchronous. Some time later, the method invoked by the callee invokes a method of the callback object letting the callee know that the method has either completed or is still working. The *monitorEvents* method requires an ACS Basic Access and Control Interface (BACI) callback string object as its parameter. The implementation of the *ACSEventAdmin* IDL interface keeps track of all callbacks that have been registered with it. Once an ALMA event is sent from any ALMA channel, *ACSEventAdmin* invokes the *working* method of all callbacks registered from *monitorEvents*. The value provided to the *working* method of the callback string object is a stringified version of the ALMA event.

Conceptually, *ACSEventAdmin* is a very powerful tool for monitoring CORBA notification channels. However, this interface exposes much of the details of the CORBA notification service to ALMA developers which violates one of the primary goals of the ACS notification channel API. Also, many ALMA developers would not use the tool if it involved writing their own CORBA client. To deal with this, a Python GUI was created which attaches itself as a CORBA client to an *ACSEventAdmin* CORBA servant. The GUI provides the core functionality of the *ACSEventAdmin* IDL interface without exposing confusing CORBA to end-users. For example, the quality of service properties used to create channels is hidden and the method used to reconfigure a channel's properties is not available.

3. CONCLUSIONS

In short, the ACS notification channel API hides some very complex CORBA code and facilitates publishing and consuming of events for ALMA developers in a simple, yet useful manner. Usage is straightforward because developers are only responsible for providing a channel name and the data to publish or a function to consume the data.

3.1. Limitations

While the ACS notification channel API has met its design requirements and goals, there are still some limitations which need to be addressed. The most notable one here is that the C++ *SimpleConsumer* class is singly templated to one *event_type*. That is, an instance of *SimpleConsumer* is only capable of receiving one type of ALMA event. This limitation is due to the fact that the *event_type* field of a *StructuredEvent* is identical to the name of the ALMA event and C++ does not support object introspection very well. The next issue to be discussed is that some qualities of service properties have not been implemented by TAO yet. TAO is an open source project which provides the C++ implementation of the CORBA notification service used by ACS. There is little that can be done here other than looking for alternative notification service implementations.

3.2. Future directions

There are three primary focus areas here and the first is hiding ALMA channels' quality of service and administrative properties within the ACS configuration database (CDB). The ACS CDB is where configuration data for all ALMA software is stored. By placing this info in the CDB, developers can use the supplier classes without ever even thinking about subclassing. This would be a much-needed improvement in the author's humble opinion. Second, ALMA events should only published when there are actually consumers subscribed for that particular type of ALMA event. This would reduce network traffic greatly. The CORBA notification service specification provides great mechanisms for doing this. Last but definitely not least, there have been some performance analyses of the ACS C++ notification channel API in the past. This was not included in this document as the information is very old and was obtained using an older, bug-prone version of the TAO notification service. The interested reader can find the related benchmarks at http://almasw.hq.eso.org/almasw/bin/view/ACS/NotificationChannelPerformances though. This information should be updated with each release cycle of ACS.

ACKNOWLEDGEMENTS

Special thanks are given to Jim Pisano of the National Radio Astronomy Observatory (NRAO) who wrote the first version of the C++ ACS notification channel API. Allen Farris, also an NRAO employee, has contributed a set of Java classes designed to simulate the behavior of CORBA notification channels within a single Java Virtual Machine (JVM) and has also had major input on exactly how much CORBA should be exposed to ALMA developers within the APIs.

Thanks are also given to Gianluca Chiozzi of the European Southern Observatory (ESO) who suggested this topic be presented at the SPIE conference and also gave valuable feedback on early drafts.

REFERENCES

1. "Dictionary.com." Dictionary.com. 2004. <http://.dictionary.reference.com> (30 April, 2004).
2. Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. New York: Addison Wesley, 1995.
3. Henning, Michi, and Steve Vinoski. Advanced CORBA Programming with C++. New York: Addison Wesley, 1995.
4. Lemke, Roland. "Notification Channel Performances." Performance of ACS 2.0 Notification Channel. January 15, 2004. <http://almasw.hq.eso.org/almasw/bin/view/ACS/NotificationChannelPerformances> (17 May, 2004).
5. Object Computing Incorporated. TAO Developer's Guide. St. Louis, MO, 1993.
6. OMG. "CORBA 3.0." Lkd. IDL Syntax and Symantics, at "formal/02-06-39." <http://www.omg.org/technology/documents/formal/corba_2.htm> (30 April, 2004).
7. OMG. "Event Service 1.1." Lkd. Event Service Specification, at "formal/2001-03-01." <http://www.omg.org/technology/documents/formal/event_service.htm> (30 April, 2004).
8. OMG. "Notification Service." Lkd. Notification Service Specification, at "formal/2002-08-04." <http://www.omg.org/technology/documents/formal/notification_service.htm> (30 April, 2004).