Version	Published	Changed By	Comment
CURRENT (v. 9)	**Jul 13, 2022 16:12**	Jorge Avarias	
v. 8	Mar 08, 2021 08:54	Jorge Avarias	
v. 7	Apr 02, 2020 16:35	Jorge Avarias	
v. 6	Apr 02, 2020 16:20	Jorge Avarias	
v. 5	Apr 02, 2020 16:11	Jorge Avarias	
v. 4	Apr 02, 2020 16:07	Jorge Avarias	
v. 3	Apr 02, 2020 15:58	Jorge Avarias	
v. 2	Apr 02, 2020 15:42	Jorge Avarias	
v. 1	Apr 02, 2020 15:41	Jorge Avarias	

This document describes C++, Python, and Java classes that provide an interface to CORBA notification channels and how to use them. Also, the design of these classes is discussed. A developer who only wishes to learn how to use these classes should read the following: **Data Definition**, **Naming Information**, **IDL Example**, and the section(s) on the specific language he or she may be interested in using.
To implement a publish-subscribe mechanism using the TAO Notification Service with structured events, there are a variety of steps that must be performed. The intent of the ACS API is to hide the details of the Notification Service – for example, the supplier and consumer proxies, notification channel factory, creating and attaching to the notification channel, publications of subscriptions and offers, etc. – and only expose the important parts which the developer must define. The developer must define data to be published on the supplier side, a function to handle incoming events for the consumer, and an IDL data structure shared by both the supplier and consumer. For the developer willing to accept default values for most things, the ACS classes provided can be instantiated as-is without the need for subclassing.
It is assumed that both the TAO notification and naming services are used for this design. The *acsStartORBSRVS* script performs this functionality although it is possible to run the services manually. Details on doing so are beyond the scope of this document though.
This design only covers a **push** notification channel whereby suppliers push data onto the notification channel.

Disclaimers

- This document does not provide detailed information on the CORBA Notify Service, the TAO group's implementation of the Notify Service, or any other CORBA service for that matter! It is only intended to discuss what ACS provides on top of these services. Please see the references section if you're interested in this type of information.
- By no means is this document intended to be used on its own! The way you'll gain the most knowledge is by looking over the examples ACS provides in the *acsexmpl*, *jcontexmpl*, and *acspyexmpl* CVS modules as well as the Doxygen and Pydoc generated documentation for the APIs in conjunction with this tutorial.
- The examples presented within this document are incomplete and have substantial blocks of code unrelated to notification channels missing. Again, please see the complete examples ACS provides in the *acsexmpl*, *jcontexmpl*, and *acspyexmpl* CVS modules.
- Regardless of what names are used within the examples, one should always follow the Software Engineering group's coding standards

Abbrevations

Acsnc	ALMA Common Software Notification Channel module
Acspy	ALMA Common Software Python module
CORBA	Common Object Request Broker Architecture
ICD Event	An ICD event is defined as the type of event sent between ALMA subsystems. Essentially this is a user-defined IDL structure and has nothing to do with CORBA structured events.
IDL	Interface Definition Language
Jcontnc	Java Container Notification Channel module
MACI	Management and Control Interfaces

NC	Notification Channel

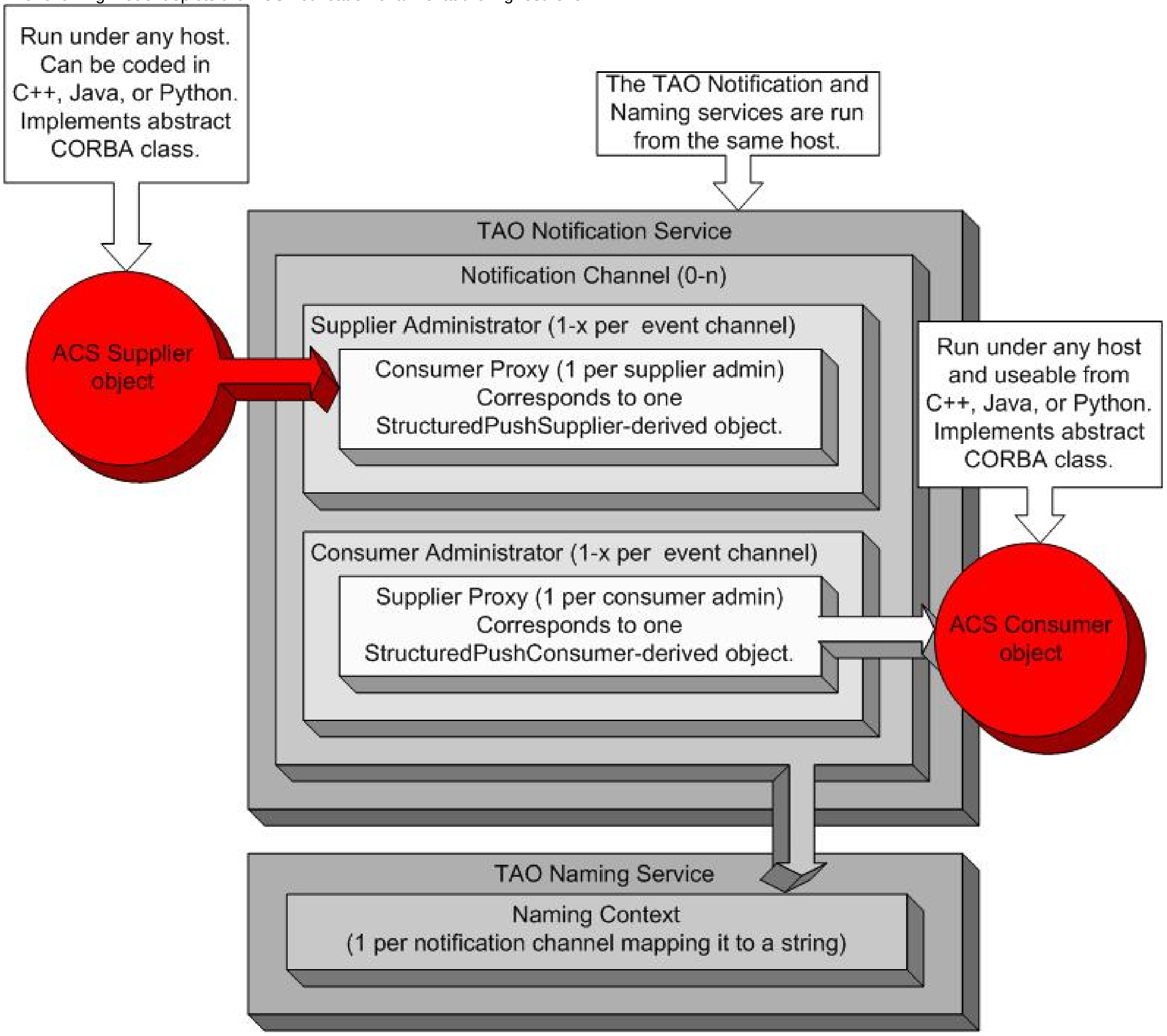
References

Requirements

- The ACS notification channel API must hide as much CORBA as possible.
- The possibility of setting the quality of service and administrative properties of channels must exist.
- Event channels should never discard events and events should be delivered to consumers in a timely manner.
- The API must be implemented in a high-performance manner to reduce the chances of events being discarded.
- As far as names for channels, domains, etc. are concerned, there must be little to no chance of suppliers and consumers confusing names.

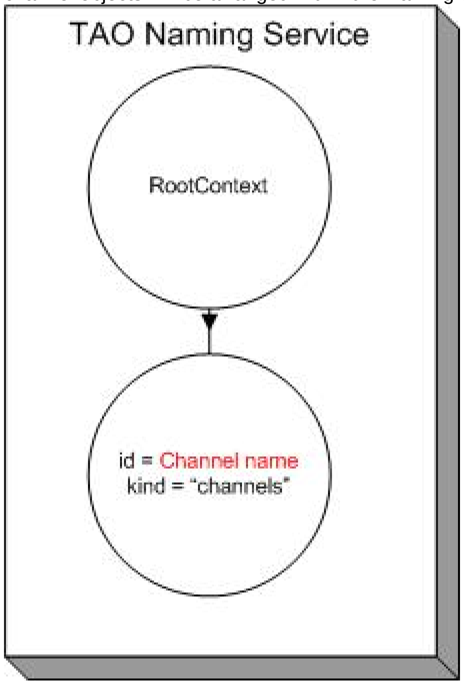
Model

The following model depicts the ACS notification channel at the highest level.


Run under any host. Can be coded in C++, Java, or Python. Implements abstract CORBA class.
The TAO Notification and Naming services are run from the same host.
TAO Notification Service
Notification Channel (0-n)
Supplier Administrator (1-x per event channel)
ACS Supplier object
Consumer Proxy (1 per supplier admin) Corresponds to one StructuredPushSupplier-derived object.
Run under any host and useable from C++, Java, or Python. Implements abstract CORBA class.
Consumer Administrator (1-x per event channel)
Supplier Proxy (1 per consumer admin) Corresponds to one StructuredPushConsumer-derived object.
ACS Consumer object
TAO Naming Service
Naming Context (1 per notification channel mapping it to a string)

Naming Service

For a consumer to subscribe to events on a given channel it first needs a reference to that channel. While there are many different ways this reference could be obtained, the Naming Service provides the safest and cleanest manner for doing so. The following diagram depicts how remote notification channel objects will be arranged within the Naming Service:



The CORBA Naming Context *id*, "Channel name", is whatever name is given to the channel by the developer. The *kind* of the Naming Context will always be "channels" differentiating event channels from other objects registered with the Naming Service.

Auto re-connection of suppliers and consumers

Auto re-connection functionality has been implemented in both suppliers and consumers in order to allow them to recreate and reconnect to the channel when the Notify Service has been restarted. This functionality is enabled by default but can be disabled using a setter method added in all respective classes.
To allow consumers to know when the Notify Service has been restarted, when a channel is created, it's registered twice in the Naming Service. The first entry is used to make publicly available the reference of the channel and the second one includes at the end of the channel's name its creation time and *kin* is set to "NCSupport". Therefore, in this way, consumers will be able to know if the Notify Service restarted by comparing the creation time registered in the Naming Service and the one they will have retrieved when they connected to the channel. When ACS launches a Notify Service it clears also all Naming Service entries that are related to this service.
The re-connection of suppliers is performed when an exception of type OBJECT_NOT_EXIST or BAD_OPERATION is thrown during the publication of an event.
Instead, re-connection of consumers is different because of their passive behavior. In this case, the implementation done adds a thread that checks the connection to the channel in a certain frequency which is 2 seconds by default. This checking process starts after waiting some time (2 seconds by default) since the last received event. Both time parameters can be changed using the respective setters added in every consumer class. Verification of the connection is performed by retrieving the channel's creation time from the Naming Service and comparing it to the creation time kept the last time the consumer connected to the channel. In case the channel's creation time couldn't be retrieved from the Naming Service, it will call the _non_existent() method of the proxy supplier in order to see if the connection is alive.
Auto re-connection of Python consumers are slightly different that the C++ and Java ones because we could reuse an already existent thread in order to perform the checking. But in this case, frequency and waiting time parameters cannot be modified so they will be always 2 seconds.
Why do we need to register the channel's creation time in the Naming Service instead of using only the _non_existent() call? This is because when the Notify Service restarts, the IDs assigned to the proxy suppliers that have been reconnected could be the same that the ones owned by suppliers that have not been reconnected so far. If this is the case, the _non_existent() call of a supplier that have not been reconnected will work because is using the same ID as one of the suppliers that have been reconnected.

ALMA Events

Within ALMA software, an event is defined to be an instance of a user-defined IDL struct. Most developers can safely skip the next section on CORBA structured events as this information is hidden within the notification channel APIs.
For those who need to know which object was responsible for sending the event, the time the event was sent, etc; take note that ACS embeds an *acsnc::EventDescription* IDL struct within the *remainder_of_body* field (see below) of CORBA structured events. See *acsnc.idl*'s Doxygen documentation for *acsnc::EventDescription* field explanations.

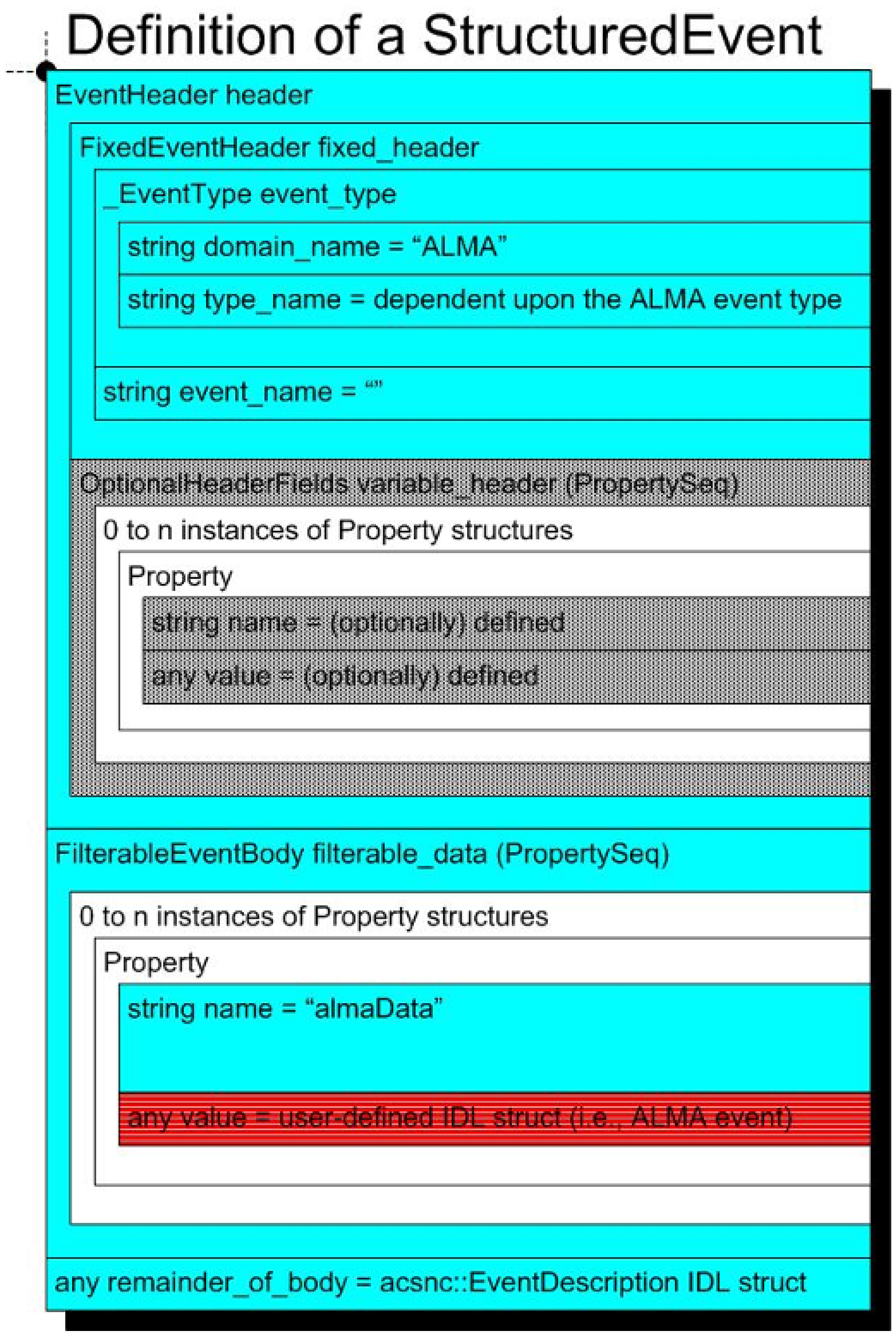
Structured Event

A structured event is the data that is transmitted from suppliers to consumers. For all intensive purposes, there are only three useful fields in this data structure: *domain_name*, *type_name*, and *filterable_data[]*.

domain_name and *type_name* are used internally within the Notification Service. For a consumer to receive an event on a given channel, it must be subscribed to that event's *domain_name* and *type_name*.

filterable_data[] is what the user needs to be concerned with. This *PropertySeq* (an array of string/CORBA Any pairs) can be filtered directly from the TAO Notification Service using Trader Constraint Language strings. A CORBA Any closely resembles a C++ void pointer and there are some stipulations on exactly what type of data can be packed into it. The data is limited to simple CORBA types (int, float, string, etc.) or some form of IDL struct/sequence. The reasoning behind this is quite simple: consumers can be written on any supported CORBA platform and some forms of data cannot be extracted (e.g, a Java consumer trying to extract a C++ void pointer is impossible). This is a restriction imposed by the CORBA specifications. In the case of IDL structs or sequences, it is necessary to compile the IDL file to obtain appropriate operators and/or helper classes to insert/extract CORBA Any data.

The other interesting fields of a structured event are: *event_name*, *variable_header*, and *remainder_of_body*. All three are optionally specified and have little to do with the way events are processed internally by the Notification Service. The developer should only be concerned with the fact that these can be used if desired.


Definition of a StructuredEvent
EventHeader header
FixedEventHeader fixed_header
_EventType event_type
string domain_name = “ALMA”
string type_name = dependent upon the ALMA event type
string event_name = “”
OptionalHeaderFields variable_header (PropertySeq)
0 to n instances of Property structures
Property
string name = (optionally) defined
any value = (optionally) defined
FilterableEventBody filterable_data (PropertySeq)
0 to n instances of Property structures
Property
string name = “almaData”
any value = user-defined IDL struct (i.e., ALMA event)
any remainder_of_body = acsnc::EventDescription IDL struct

Quality of Service and Administrative properties are what sets the Notification Service apart from the CORBA Event Service. The Notification Service specification describes various properties which control how events, suppliers, consumers, and even channels behave under various circumstances.
For your convenience, all properties are described in detail. Please note that **some of these properties have not yet been implemented** by TAO though (see the *Known Problems* section).\

Quality of Service

EventReliability	Handles the kind of guarantee that can be given for a specific event getting delivered to a *Consumer*. Represented by *BestEffort* or *Persistent*.
ConnectionReliability	Handles the kind of guarantee that can be given for the connections between the Notification Channel and its clients. Can have the same values as *EventReliability.*
StopTime	Specifies an absolute expiry date.
Timeout	Specifies a relative expiry date.
StartTime	Specifies an absolute time after which the event will be discarded.
PriorityOrder	Used to control the order of the events delivered to the consumers. Default value is 0 but ranges from –32767 to 32767.
OrderPolicy	Used by a proxy to arrange events in the dispatch queue. Can be the following values: *AnyOrder*, *FifoOrder*, *PriorityOrder*, and *DeadlineOrder*.
DiscardPolicy	Defines the order in which events are discarded when overflow of internal buffers occur. Holds the same values as *OrderPolicy* plus an addition value: *LifoOrder*.
MaximumEventsPerConsumer	Defines a bound for the maximum number of events the channel will queue for a given consumer. The default value is 0.
MaximumBatchSize	Irrelevant to the ACS API!
PacingInterval	Irrelevant to the ACS API!

Administrative

MaxQueueLength	Specifies the maximum number of events the channel will queue internally before starting to discard events.
MaxConsumers	Defines the maximum number of consumers that can be connected to a particular channel.
MaxSuppliers	Defines the maximum number of suppliers that can be connected to a particular channel.
RejectNewEvents	Defines how to handle events when the number of events exceeds *MaxQueueLength* value. Set this *Boolean* value to *true* for *IMPL_LIMIT* exceptions to be thrown. A *false* value implies events will be discarded silently.

Both the supplier and consumer must have agreed-upon names for the notification channel. This section describes how suppliers and consumers discover the notification channel to use.
There are three important names:

- Channel Name – the name of the channel created by a supplier or subscribed to by a consumer. This name is registered with the Naming Service. The developer sets this via supplier/consumer constructor parameters.
- Event Domain – "ALMA". Directly corresponds to the *domain_name* field in a structured event. Hidden within the APIs.
- Event Type – identifies the type of an (ALMA) event. Corresponds to the *type_name* field in a structured event. Hidden within the APIs which automatically determines this name from the name of the ALMA event being published.

At present, there are two recommendations regarding names usage:

- Channel names should be specified as constant strings in IDL. This is done to enforce agreed-upon names between suppliers and consumers.
- Prepending the subsystem name to a given channel name reduces the chances of name conflicts between subsystems. An example would be the Correlator subsystem creates the "CORRdata" channel instead of just "data".

Supplier (C++ and Python)

This class provides the interface used to push CORBA structured events onto the notification channel. It creates the channel (if it doesn't exist) and hides most CORBA from the developer.

SimpleSupplier (C++ and Java)

SimpleSupplier is a subclass of *Supplier* designed specifically for publishing ALMA events. It provides the interface used to push IDL structs onto the notification channel, creates the channel (if it doesn't exist), and hides all CORBA from the developer. In Python, the *Supplier* class performs the same functionality as *SimpleSupplier*.

RTSupplier (C++)

RTSupplier, a subclass of *Supplier*, is basically identical to *SimpleSupplier* except that it is to be used in real-time applications. In a nutshell, the publish method queues events locally and a low-priority thread publishes events across the network.

Consumer

This class provides the interface used to subscribe to CORBA structured events from the notification channel. It is used by the data consumer and must be derived to overload *push_structured_event* (in C++) or *processEvent* (in Java and Python) to get the requested data. It should be noted that *Consumer* does not have to be subclassed in Java/Python because those implementations use handler functions and receiver objects. *Consumer* will create the channel if it does not already exist.

SimpleConsumer (C++)

SimpleConsumer is a concrete, templated implementation of *Consumer* which uses a single handler function to process a single type of event. There is no need to subclass *SimpleConsumer* and the Java and Python implementations of *Consumer* include identical functionality.

The following excerpt from an IDL file defines channel names, struct definitions, etc. that will be used for examples throughout this document. Please note that items in **bold** should be adapted for your applications:

ACS/LGPL/CommonSoftware/acsexmpl/ws/idl/acsexmplFridge.midl

```
#pragma prefix "alma"
module FRIDGE
{
enum TemperatureStatus { OVERREF, ATREF, BELOWREF};

struct temperatureDataBlockEvent
{
        float absoluteDiff;
        TemperatureStatus status;
};

const string CHANNELNAME_FRIDGE = "fridge";
```

4	The actual temperature of the fridge can be over/at/below the desired temperature.
6-10	This defines the block of data that will be sent over the notification channel.
12	The channel name for use with all fridge suppliers and consumers.

In this tutorial, a *SimpleSupplier* example which publishes one event and then disconnects from the channel is given. For specifics on *Supplier* functions, please see the Doxygen-generated documentation of the *acsnc* module. Aside from that, C++ Supplier objects need to be instantiated from within the context of a component or after a MACI SimpleClient has been created.

Example

Bolded code should be adapted for the developer's particular needs. Note that all component code has been omitted and the *acsexmpl* module contains the example in its entirety.
ACS/LGPL/CommonSoftware/acsexmpl/ws/src/acsexmplFridgeImpl.cpp:

ACS/LGPL/CommonSoftware/acsexmpl/ws/src/acsexmplFridgeImpl.cpp

```
#include <acsexmplFridgeS.h>
#include <acsncSimpleSupplier.h>

nc::SimpleSupplier {}m_FridgeSupplier_p* = 0;

m_FridgeSupplier_p = new nc::SimpleSupplier(FRIDGE::CHANNELNAME_FRIDGE,
                                                                                                   this);

FRIDGE::temperatureDataBlockEvent t_data;
t_data.absoluteDiff = (double)0.0;
t_data.status = FRIDGE::ATREF;

m_FridgeSupplier_p->publishData<FRIDGE::temperatureDataBlockEvent>(t_data);

if (m_FridgeSupplier_p)
{
        m_FridgeSupplier_p->disconnect();
        m_FridgeSupplier_p=0;
}
```

1	A user-defined IDL structure is packed into all events. Therefore, the stub header must be included.
4	Declare a pointer for the *SimpleSupplier* class.
6-7	Create and initialize the *SimpleSupplier* instance.
6	The channel events will be published on.
7	A pointer to an *ACSComponentImpl* is needed to pack the component's name into the event.
9	Create a new **temperatureDataBlockEvent** that will be packed into an event.
10-11	Fill the fields of the **temperatureDataBlockEvent** with arbitrary data.
13	*publishData* is a blocking call that actually sends the structured event to consumers. The template parameter is identical to the type of data being published.
15-19	It is necessary to *disconnect* from the channel after we are done sending events to prevent remote memory leaks from occurring. One should never delete the *Supplier* directly as it's a CORBA object.

A C++ *SimpleConsumer* utilizes a handler function for a single event type and does not have to be overridden. C++ Consumer objects need to be instantiated from within the context of a component or after a MACI SimpleClient has been created.

Example

The following depicts the usage of a *SimpleConsumer*. As always, **bolded** text should be adapted for your particular needs and error handling has been omitted.

ACS/LGPL/CommonSoftware/acsexmpl/ws/src/acsexmplClientFridgeNC.cpp

```
#include "acsexmplFridgeC.h"
//…
//------------------------------------------------------------------------
void myHandlerFunction(FRIDGE::temperatureDataBlockEvent joe, void *other)
{
    ACS_SHORT_LOG((LM_INFO, "::myHandlerFunction(...): %f is the tempdiff.",
    joe.absoluteDiff));
}
//------------------------------------------------------------------------
//…

nc::SimpleConsumer<FRIDGE::temperatureDataBlockEvent> simpConsumer_p* = 0;
ACS_NEW_SIMPLE_CONSUMER(simpConsumer_p,
                                                FRIDGE::temperatureDataBlockEvent,
                                                FRIDGE::CHANNELNAME_FRIDGE,
                                                myHandlerFunction,
                                                (void *)0);
simpConsumer_p->consumerReady();

ACE_Time_Value time(150);
client.run(time);
simpConsumer_p->disconnect(); simpConsumer_p=0;

//…
```

1	The client header stub generated by the IDL compiler is used to insert and extract user-defined IDL structures to and from CORBA Any's.
4-8	*myHandlerFunction* is a void function designed to manipulate one type of IDL struct/event, a *temperatureDataBlockEvent*. Each time a *temperature DataBlockEvent* event is received, *SimpleConsumer* will call this function.
4	Handler functions must not return any value and shall take in two parameters: the user-defined IDL structure defining ICD events and a void * whose value will always be identical to the void * on line 17.
6-7	Do something useful with the event.
12-17	A *SimpleConsumer* instance is created. Note that the *ACS_NEW_SIMPLE_CONSUMER* macro must be used to do this!
12	Create a *SimpleConsumer* pointer using the event type *FRIDGE::temperatureDataBlockEvent* for the templated parameter.
13	*simpConsumer_p* is a pointer to an unallocated *SimpleConsumer*.
14	The second parameter is the type of event to be received (i.e., a user-defined IDL struct). *SimpleConsumers* are only capable of receiving and processing a single type.
15	The name of the channel we will subscribe too. In this case it's the "fridge" channel.
16	This function will be invoked each time an event is received.
17	A void * that will be sent to the handler function (line 4) each time an event is received. It can very useful to pass in an object for the void * which will perform some operation on the event from the handler function.
18	Tell the channel we are ready to begin consuming events.
20-22	Run this example for 150 seconds and then disconnect the consumer from the channel.
22	Disconnect from the channel to prevent remote memory leaks from occurring. Do not delete the *Consumer*!

Python suppliers are very powerful in the fact that they do not have to be subclassed and act as *SimpleSupplier*'s.

Example

The following depicts the full implementation of a *Supplier* which publishes one event and then disconnects from the channel. As always, **bolded** text should be adapted for your particular needs and error handling has been omitted.

ACS/LGPL/CommonSoftware/acspyexmpl/src/acspyexmplFridgeNCSupplier.py

```
#!/usr/bin/env python
#------------------------------------------------------------------------------
import FRIDGE
from Acspy.Nc.Supplier import Supplier
#------------------------------------------------------------------------------
g = Supplier(FRIDGE.CHANNELNAME_FRIDGE)
h = FRIDGE.temperatureDataBlockEvent(3.7, FRIDGE.ATREF)
g.publishEvent(h)
g.disconnect()
#------------------------------------------------------------------------------
```

3	*FRIDGE* is the CORBA stub module generated by the IDL compiler. It contains the user-defined IDL structure this script is designed to publish as well as the channel's name.
6	This supplier will publish events to the "fridge" channel. Within a component, a second parameter (i.e., *self*) would also have to be passed.
7	Create an instance of the user-defined IDL structure to be published.
8	Publish the IDL structure created on line 7.
9	Disconnect from the notification channel.

A Python consumer is by far the easiest to use of the ACS-supported CORBA language mappings.

Example

The following depicts the full implementation of a *Consumer*. As always, **bolded** text should be adapted for your particular needs and error handling has been omitted.
ACS/LGPL/CommonSoftware/acspyexmpl/src/acspyexmplFridgeNCConsumer.py

ACS/LGPL/CommonSoftware/acspyexmpl/src/acspyexmplFridgeNCConsumer.py

```
#!/usr/bin/env python
from time import sleep
import FRIDGE
from Acspy.Nc.Consumer import Consumer
#------------------------------------------------------------------------------
def fridgeDataHandler(some_param):
    tempDiff = some_param.absoluteDiff
    print 'The temperature difference is', tempDiff
    return
#------------------------------------------------------------------------------
g = Consumer(FRIDGE.CHANNELNAME_FRIDGE)
g.addSubscription(FRIDGE.temperatureDataBlockEvent, fridgeDataHandler)
g.consumerReady()
sleep(50)
g.disconnect()
#----------------------------------------------------------------------------
```

3	*FRIDGE* is the CORBA stub module generated by the IDL compiler. It contains the user-defined IDL structure this script is designed to process as well as the channel's name.
6-9	We must define a function which is capable of manipulating *FRIDGE.temperatureDataBlockEvent*'s. This function will be invoked each time an event is received from the notification channel by registering it with the consumer (line 12). Take note that *someParam* will always be an instance of *FRIDGE.temperatureDataBlockEvent*.
11	Create an instance of *Consumer* connecting to the "fridge" channel.
12	Subscribe to *FRIDGE.temperatureDataBlockEvent* events and inform the consumer it needs to invoke *fridgeDataHandler* each time a *temperatureDataBlockEvent* event is received.

13	Let the channel know we are ready to start processing events.
14	Give suppliers time to publish events…
15	Disconnect from the channel.

Java suppliers and consumer need to be run from the context of a component or ACS Java client.

Example

The following shows an example publishing one event and then disconnecting from the channel. **Bold text** needs to be adapted for your particular needs and error handling has been omitted.

ACS/LGPL/CommonSoftware/jcontexmpl/src/alma/demo/EventSupplierImpl/EventSupplierImpl.java

```
//…
import alma.acs.nc.SimpleSupplier;
import alma.FRIDGE.temperatureDataBlockEvent;
import alma.FRIDGE.TemperatureStatus;
//…
SimpleSupplier m_supplier = null;
m_supplier = new SimpleSupplier(alma.FRIDGE.CHANNELNAME_FRIDGE.value, m_containerServices);

temperatureDataBlockEvent t_block = new temperatureDataBlockEvent(3.14F,

TemperatureStatus.ATREF);

m_supplier.publishEvent(t_block);
m_supplier.disconnect();
```

1 & 5	Normally suppliers will be contained within components. This example assumes nothing about whether it's operating under the context of a component, client, etc.
2	Import the *SimpleSupplier* class used to publish events.
3-4	CORBA Stub classes generated by the IDL to Java compiler. These define the event to be published.
6	The *SimpleSupplier* variable to be used.
7	The first parameter of *SimpleSupplier*'s constructor is the name of the channel events will be published on. A *ContainerServices* object is required as the second parameter to gain access to the ORB.
9-10	Create an instance of the IDL structure to publish.
12	Publish the event.
13	Disconnect from the channel.

Example

What follows is a trivial consumer that processes one event and then disconnects from the channel. ACS/LGPL/CommonSoftware/jcontexmpl/src/alma/demo/EventConsumerImpl/EventConsumerImpl.java:

ACS/LGPL/CommonSoftware/jcontexmpl/src/alma/demo/EventConsumerImpl/EventConsumerImpl.java

```
import alma.acs.nc.Consumer;
//…
private Consumer m_consumer = null;
//…
public void receive(alma.FRIDGE.temperatureDataBlockEvent joe)
{
        System.out.println("The temp difference is:" + joe.absoluteDiff);
}
//…
m_consumer = new Consumer(alma.FRIDGE.CHANNELNAME_FRIDGE.value, m_containerServices);
m_consumer.addSubscription(alma.FRIDGE.temperatureDataBlockEvent.class, this);
m_consumer.consumerReady();
m_consumer.disconnect();
```

1	The location of the *Consumer* class in Java.
5-8	We define a "receive" method designed to process the particular type of event we are interested in. Please note that it must be named "receive" and it does not matter which class this method is implemented in. In this example, it was defined in the component implementation to keep things simple.
10-11	The constructor must subscribe to a specific channel providing a reference to a *ContainerServices* object (line 10) and then add a subscription for a specific type of event (line 11). On line 11, the 2nd parameter to *addSubscription* must be an object implementing *receive (temperatureDataBlockEvent someEvent)*.
12	*consumerReady* must be invoked to start receiving events.
13	Failure to disconnect from the channel causes remote memory leaks!

The following describes a set of *Supplier* and *Consumer* subclasses contributed by the Scheduling subsystem designed primarily for simulating CORBA Notification Channels entirely within Java virtual machines.

Package/Namespace

alma.acs.nc

Terminology

Publisher is equivalent to *Supplier*.
Receiver is equivalent to *Consumer*.

Classes

- **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 3.0 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 3.0 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.

The Local Notification Channel

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.
NOTE: For the local notification channel your application has to be running on one JVM. Otherwise channels will not find each other!

Do they work with Python and C++?

Yes! The examples below have been tested with Python and C++ suppliers. Likewise, FridgePublisher has been tested with a python consumer and a C++ consumer.

Examples

CORBA Publisher

- See ACS/LGPL/CommonSoftware/jcontnc/test/alma/demo/test/AbstractNC/NCPublisherImpl.java

CORBA Receiver

- See ACS/LGPL/CommonSoftware/jcontnc/test/alma/demo/test/AbstractNC/NCReceiverImpl.java

Local Publisher

- See ACS/LGPL/CommonSoftware/jcontnc/test/alma/demo/test/LocalNC/TestLocalNC.java

Local Receiver

- See ACS/LGPL/CommonSoftware/jcontnc/test/alma/demo/test/TestNCReceiver/TestReceiver.java

Channel Properties and the ACS CDB

When using the ACS Notification Channel framework, there is nothing special required from the developer aside from issuing the usual ACS startup command(s) which in turn start the CORBA services. However, for those wishing to modify the Quality of Service and Administrative properties for a channel, they can do the following:

1. For a given channel, "xyz", create $ACS_CDB/CDB/MACI/Channels/xyz/xyz.xml which validates against $ACSROOT/config/CDB/schemas /EventChannel.xsd. The EventChannel.xsd XML schema defines all Quality of Service and Admin properties (described in the Notification Serverice Properties section of this document) applicable to the type of channels the ACS API creates.
2. Start (or restart) all of ACS.
3. The first time a supplier or consumer object tries to access the "xyz" channel it will be created using the properties you specified in the ACS CDB. The only special thing to note here is that because the implementation of the Notification Service we use does not support all properties, the API may reject some of your specifications although no exceptions will be thrown. See the known problems sections for details.

Debugging Functionality

Quite often it seems to be the case that there is confusion about whether events have been received or even sent for that matter. In as such, the *EventChannel.xsd* schema now provides an extra attribute, *IntegrationLogs*, having nothing to do with properties of the channel. When this Boolean value is set to true, a log is sent out each time an event is published or consumed. Be careful though as logs are also events and using this mechanism instantly doubles the overhead being placed on the CORBA Notification Service. It is likely that this debugging functionality will be removed to improve performance someday but for the time being it should prove to be an invaluable too.

Event Handler Timeouts

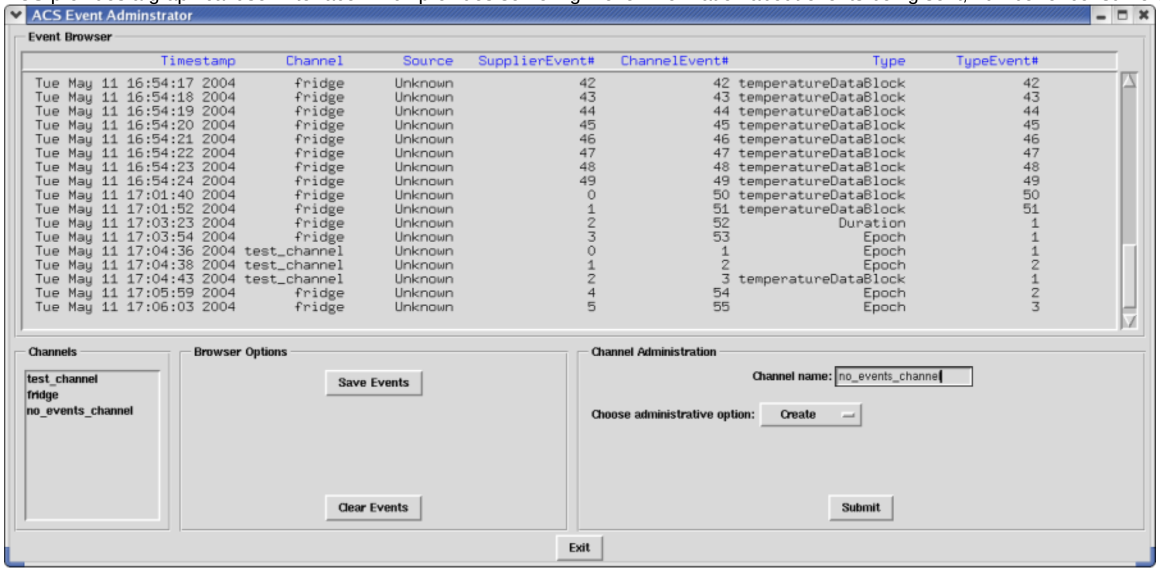
Overloading the CORBA notification service with enormous amounts of events and consumer objects that take too long processing the events can cause serious performance issues or even the corruption of the notification service process itself. To help deal with this, ACS has introduced a new optional sequence of XML elements, *Events:, within the main _EventChannel* element of *EventChannel.xsd*. By setting the "MaxProcessTime" attribute of this new element, one can make the ACS NC API(s) emit warning messages whenever an event handler method or function takes too long processing an event. Even if you choose not to use the CDB to set this, by default these messages are sent if the handler takes more than two seconds to process an event.
As an all too brief example, let's say you are the developer sending the events and know the frequency of a particular type of event, IDL:/alma/FRIDGE /FridgeTemperatureDataBlockEvent:1.0, will be shorter than the two second maximum ACS automatically sets. Assume this frequency is once per second. It should then be obvious that any consumers receiving these events need to process them within one second or risk having the event being added to a queue somewhere thereby eventually eating up all available memory. What you can do to be made aware of this problem at run-time is to add the following XML element to the XML described in the section 15.1:

```
<Events>
<_ Name="FridgeTemperatureDataBlockEvent" MaxProcessTime="1.0"/>
</Events>
```

The only other piece of advice is that any number of these "_" elements can appear within the "Events" element.

Event Browser

ACS provides a graphical user interface which provides some high-level information about events being sent, number of consumers, etc.:



Timestamp	Channel	Source	SupplierEvent#	ChannelEvent#	Type	TypeEvent#
Tue May 11 16:54:17 2004	fridge	Unknown	42	42	temperatureDataBlock	42
Tue May 11 16:54:18 2004	fridge	Unknown	43	43	temperatureDataBlock	43
Tue May 11 16:54:19 2004	fridge	Unknown	44	44	temperatureDataBlock	44
Tue May 11 16:54:20 2004	fridge	Unknown	45	45	temperatureDataBlock	45
Tue May 11 16:54:21 2004	fridge	Unknown	46	46	temperatureDataBlock	46
Tue May 11 16:54:22 2004	fridge	Unknown	47	47	temperatureDataBlock	47
Tue May 11 16:54:23 2004	fridge	Unknown	48	48	temperatureDataBlock	48
Tue May 11 16:54:24 2004	fridge	Unknown	49	49	temperatureDataBlock	49
Tue May 11 17:01:40 2004	fridge	Unknown	0	50	temperatureDataBlock	50
Tue May 11 17:01:52 2004	fridge	Unknown	1	51	temperatureDataBlock	51
Tue May 11 17:03:23 2004	fridge	Unknown	2	52	Duration	1
Tue May 11 17:03:54 2004	fridge	Unknown	3	53	Epoch	1
Tue May 11 17:04:36 2004	test_channel	Unknown	0	1	Epoch	1
Tue May 11 17:04:38 2004	test_channel	Unknown	1	2	Epoch	2
Tue May 11 17:04:43 2004	test_channel	Unknown	2	3	temperatureDataBlock	1
Tue May 11 17:05:59 2004	fridge	Unknown	4	54	Epoch	2
Tue May 11 17:06:03 2004	fridge	Unknown	5	55	Epoch	3

Channels: test_channel, fridge, no_events_channel

Channel name: no_events_channel

Choose administrative option: Create

This GUI is started by running "acseventbrowser" from the command-line after Manager is up and running.

- **Not all Quality of Service and Administrative Properties Work:** This is the price we pay for using free software. Specifically the following have not been implemented by TAO: *EventReliability*, *ConnectionReliability*, *StopTime*, *StartTime*, and *PriorityOrder*.
- **Event Browser does not see all events**: The cause of this that we've seen is that when supplying events, developers forget to set fields of the IDL structure (particularly enumeration fields). There is an SPR on this and we've submitted bug reports to the various ORB vendors. No progress seems to have been made at this point.

- **Event Filtering does not work on so-called "ALMA events"**: The TAO Notify Service does not support event filtering on user-defined IDL structures even though the CORBA Extended Trader Constraint Language (a.k.a. event filtering language) specifies it should.

- **For the most up-to-date information on known Notification Channel problems, check the ALMA Software Engineering Software Problem Report (SPR) system.**

Your best source of information is from the code itself (i.e., Doxygen). Other than that, here are the locations (in CVS) of all examples in this document:
ACS/LGPL/CommonSoftware/acsexmpl/ws/idl/acsexmplFridge.midl
ACS/LGPL/CommonSoftware/acsexmpl/ws/include/acsexmplFridgeImpl.h
ACS/LGPL/CommonSoftware/acsexmpl/ws/src/acsexmplFridgeImpl.cpp
ACS/LGPL/CommonSoftware/acsexmpl/ws/src/acsexmplClientFridgeNC.cpp
ACS/LGPL/CommonSoftware/acspyexmpl/src/*.py
ACS/LGPL/CommonSoftware/jcontexmpl/src/alma/demo/Event*
Links to the Doxygen, Javadoc, and Pydoc generated documentation can be found on the ACS webpage.