Borland VisiBroker™ 8.0

VisiNotify Guide

Borland Software Corporation
20450 Stevens Creek Blvd., Suite 800
Cupertino, CA 95014 USA
www.borland.com

Refer to the file deploy.html for a complete list of files that you can distribute in accordance with the License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

VB 80 VisiNotify Guide
April 2007

Borland®

Contents

1

Introduction to VisiNotify

This chapter provides a general discussion on the architecture of OMG Event/ Notification Service and introduces Borland's implementation, VisiNotify.

Note

It is recommended that you use this document in conjunction with the application examples shipped with VisiNotify and the OMG Event/Notification Specification that is available in the OMG web site.

OMG Event/Notification Service Communication Model

In the CORBA environment, the core ORB is a distribute framework for creating object-oriented client/server applications. The communication model(s) supported by the core ORB is intended for the client/server applications with direct (at least conceptually), one-to-one, synchronous communication. Some of the application requirements go beyond the core ORB facility, such as:

- Support for distributed publish/subscribe application design, such as, many-to-many and decoupled.
- Support for single directional, asynchronous and buffered event distribution with a throughput substantially higher than synchronous communication.
- Support for quality of services (QoS), such as event/connection reliability.
- Support for event filtering.

The requirements mentioned above have been supported by traditional message oriented middleware (MOM) for non-CORBA applications. OMG Event/Notification Service addresses the same set of requirements for CORBA applications.

In **publish/subscribe** applications, objects involved in the communication could be arbitrary. There are two types of objects in publish/subscribe communication; the event suppliers (providers and publishers) and event consumers (observers and subscribers). Also, there are two event transfer models; the event pushing and event pulling. Objects involved in the publish/subscribe communication are decoupled from each other by the message middleware. These objects are not dependent on the existence and status of other objects in order to work properly. Event suppliers transfer events to the channel regardless the existence of consumers.

Note

Decoupling in this instance means independence rather than security. If a supplier can tell, implicitly, the existence of a consumer, it does not mean decoupling is broken.

In **single directional** event distribution, events flow from upstream into downstream. Specifically, events flow from suppliers to channels and subsequently flow from channel into subscribed consumers. Event transfer is **asynchronous** and buffered. Suppliers can get acknowledgement from the message middleware, not from the consumers. This means, event transfer routing through a message middleware could have much higher throughput than synchronous method invocation without routing.

OMG Event/Notification Service Object Model

The main concept in OMG Event/Notification is the **channel**. Events are sent into an event channel and replicated to their recipients. Multiple independent channels can be created and used by a given application. Events are either pushed or pulled into an event channel from supplier. The events flow inside the channel in a downstream direction. Events in the downstream end are buffered in proxy suppliers and are pushed to or pulled by consumers. Application level event suppliers or consumers are connected with proxy objects to transfer events into/from the channel.

In the **downstream** end of a channel (consumer end):

- Each push consumer needs to create and connect to a dedicated proxy push supplier. It then passively waits for the channel callback to send events.
- Pull consumers actively invokes request on proxy pull suppliers to retrieve events from the channel.

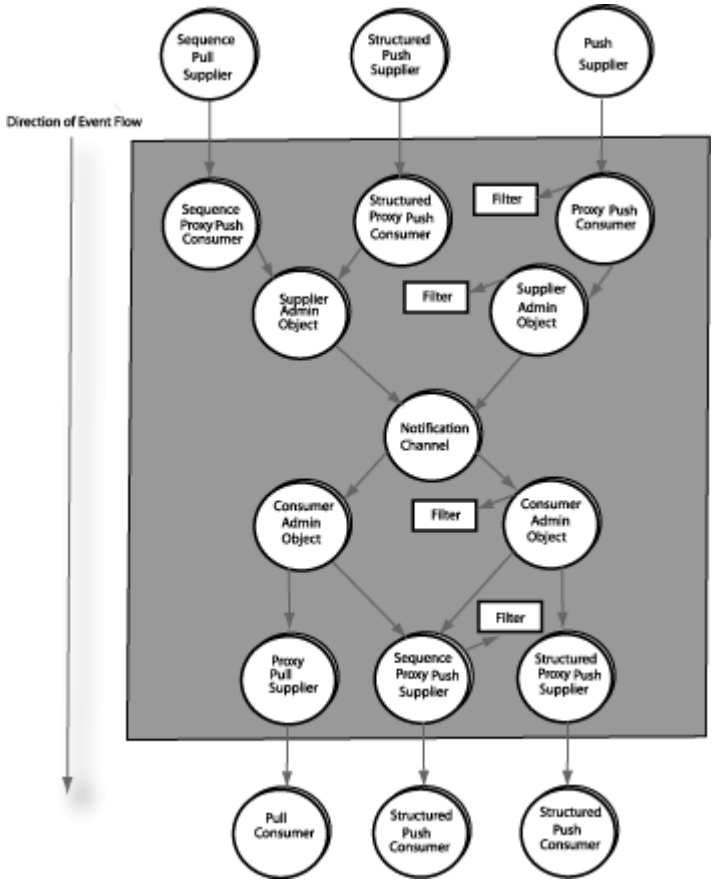
Proxy suppliers are usually located in the channel server and are created by applications from consumer admins. Consumer admins are created either as a default or by applications from channels. Each channel has a default consumer admin. This creation process forms a [channel]–[consumer(s) admin(s)]–[supplier proxy(s)] hierarchy.

In the **upstream** end of a channel (supplier end):

- Push suppliers actively invoke requests on proxy push consumer to push events into the channel.
- Each pull supplier needs to create and connect to a dedicated proxy pull consumer. It then passively waits for channel callback to retrieve events.

Proxy consumers are usually located in the channel server and are created by applications from supplier admins. Supplier admins are created either as a default or by applications from channels. Each channel has a default supplier admin. This creation process forms a [channel]–[supplier(s) admin(s)]–[consumer proxy(s)] hierarchy.

This diagram illustrates the event flow (upstream and downstream) within the notification communication model:



Like most traditional message oriented middleware, the OMG Notification Service also defines and supports Qualify of Services (QoS). VisiNotify supports most OMG defined QoS policies along with additional VisiNotify extensions. Among those QoS policies, two most important QoSs are event persistence and connection persistence. With event persistence (or reliability), buffered events in the channel are temporarily stored in persistent repository to prevent event loss due to maintenance shutdown or accident system crash. With connection persistence (or reliability), OMG defines two QoS functions. The first function is, images of channels, admins, proxies and their current settings are stored in a persistent repository that allows the channel server to restore these objects upon channel restart. The second function is, the channel can reestablish transport connections to pull suppliers and push consumers.

Another important service defined by the OMG Event/Notification Service is event filtering. Applications can add filter objects at admin or proxy level to selectively filter out unwanted events.

The OMG Event/Notification Service defines four types of notification channels; the **untyped**, **structured**, **sequence**, and **typed**. The event interfaces of the first three channels are predefined by OMG Event/Notification specification and are referred as “predefined” channel. The event interfaces for typed channels are not predefined by OMG but by user applications and are referred as “user defined” typed channel. VisiNotify supports all four types of channels with the except of the sequence pulling.

With the untyped channel, events are represented as CORBA *Anys*. Events are sent by invoking push() operation with CORBA::Any as an input parameter using untyped consumer or proxy untyped consumer objects. With structured and sequence channel, events are represented as StructuredEvent IDL structures or sequence. Events are sent by invoking push_structured_event() or push_structured_events() on respective consumer or proxy consumer. With typed channel, there is no predefined event interface. Event interfaces are defined by user applications as OMG IDL interfaces. Events are sent by invoking non-pseudo operations on consumers or proxy consumers typed interfaces.

Note

Examples of supplier and consumer applications with above event types are discussed in chapter “Developing Supplier and Consumer Applications” of this document.

VisiNotify features

Borland VisiNotify is an industrial strength implementation of OMG Event/Notification Service. Instead of on implementing on the user level, VisiNotify is implemented on ORB level and registered with the Naming Service using the generic Naming Service mechanism. See Using the VisiNaming Service (Java) and Using the VisiNaming Service (C++) for more information. With this unique design, VisiNotify is able to work more efficiently and to provide features that are difficult or impossible to support on the user level. Here are the main features of VisiNotify:

Superior throughput and scalability

VisiNotify is designed to work at the GIOP message level. It directly hands over received event payloads to the downstream consumers. When replicating any received events, VisiNotify does not de-marshal events unless there are filters or event level QoS in the stream. And VisiNotify does not re-marshal events even if there are filters or event level QoS. This unique design allows VisiNotify to reach a very high event throughput with a very low CPU usage. On handling client connect through GIOP 1.0 and 1.1, a series advanced techniques are used to adjust payload alignment without de-marshalling and re-marshalling the events.

By leveraging Borland's event buffering/batch technology in VisiBroker 5.1, the throughput displayed by VisiNotify is substantially higher in magnitude than any user level notification service product on the market. Event buffering/batch optimizes VisiNotify throughput. Different from user level batch technology, (such as sequence event) the event buffering/batch technology is fully transparent to user applications and has no restrictions on event type. All event types (untyped, structured, sequence or typed) can be buffered/batched. Therefore, VisiNotify is able to reach the best end-to-end event throughput by combining event batch with the smallest event sizes and lowest event marshalling/de-marshalling cost of typed events.

With user level implementation, event buffering/batch is not transparent to application. Also, only restricted event type, namely structured event, can be sent in batches. Comparing to Borland's event buffering/batch technology, event batch using sequence channel has no advantages. Therefore, VisiNotify only provides limited support for sequence channel with the following restrictions:

- Support for only end-to-end push model sequence channel. Filter constraints and event level QoS policies are only evaluated for the first event in a sequence and the result is applied to the entire event.
- Maximum batch size setting is ignored.
- Sequence pulling is not supported.

Note

A real industry usage case (ITU–T CORBA/TMN notification) based throughput benchmark test suit is shipped with this VisiNotify release (examples/vbroker/notify/bench_cpp and bench_java).

Superior performance with event persistence

Many user level channel products use DynAny to unpack event from events for persistence support. VisiNotify directly dumps event message payloads into persistent storage without unmarshalling and unpacking them. This unique design minimizes the overhead from event persistence. Under the default setting, VisiNotify event persistence overhead is 5% to 15%.

Valuetype support

VisiNotify is the first and only notification channel that supports valuetypes in events. Even with the presence of a filter in the event stream, VisiNotify can still evaluate filter conditions using the attributes before the first valuetype in a given event.

Typed channel support

The typed channel support is documented in the Developing Supplier and Consumer Application chapter.

VisiNotify is the first OMG Typed Event/Notification implementation that does not use Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI). VisiNotify does not rely on Interface Repository to work unless there is a filter constraint in the typed event stream. These means, VisiNotify's typed channel is significantly faster than any typed or untyped channel implementations.

Since VisiNotify does not rely on the IR when filter is not used, the key parameters used in calling obtain_typed_..._consumer/supplier() are not necessarily to be the event interface repository id. Therefore, applications can choose the proxy keys as an alternative filtering strategy. Application can use proxy keys to divide a given typed channel into multiple logical channels. This approach is more efficient and flexible than the constraint language parsing based filtering.

Publish/Subscribe Adapter (PSA)

The Publish/Subscribe Adapter feature is documented in Publish/Subscribe Adapter (PSA) chapter.

The PSA is a programming model and software component supported by VisiBroker 5.1. It works on top of any OMG Event/Notification Service. The basic concept of the PSA is to provide a high level object-oriented abstraction for publish/subscribe communication. The PSA not only simplifies the code of typed event/notification applications and provides an elegant solution for typed pulling, it also shields the application from directly dealing with the connection interface difference. Without PSA, using different event types (untyped, structured, sequence and typed) or transfer models means different connection interfaces.

Typed pulling without using Pull<I> interface

One elegant feature of the PSA is that it supports typed pulling using the original user defined <I> interface instead of the mangled Pull<I> interface.

Explicit RMI and EJB support

The explicit RMI and EJB support is documented in the Developing Supplier and Consumer Application chapter.

VisiNotify supports two types of RMI/EJB connection scenarios. The first scenario is the typed event RMI/EJB applications using VisiNotify's typed channel as a message middleware. In this case, user defined RMI interface or EJB remote interface is the definition for the typed event interface. All suppliers are RMI applications making RMI call to push events into VisiNotify typed channel. All connected consumers are also RMI applications with their RMI reference connected to the typed event channel.

The second scenario involves using the structured event channel. In this scenario, all suppliers are CORBA applications sending CosNotification::StructuredEvent to structured event channel. In the downstream end, some consumer applications can be a CORBA application connected as structured consumers while others are consumers that can be structured event EJB beans. A structured event EJB bean is not different from normal session or entity bean. A structured event bean and its remote interface implements and declares a push_structured_event() operation with org.omg.CosNotification.StructuredEvent as input parameter. VisiNotify provides an utility, **subtool**, to connect a structured event bean's remote interface to a given VisiNotify structure event channel.

These two scenarios provide alternative and pure object-oriented solutions for event driven J2EE applications. Comparing to Java Message Service (JMS) and Message Driven Bean (MDB), the advantages are:

- Static type safe RMI stubs and skeleton perform message pack/unpack
- Event are described by user defined Java RMI interface.

Connection persistence

VisiNotify supports the connection persistence as defined by OMG specification:

- Restore persistent channels, admins and proxies after channel restart.
- Reestablish broken/lost transport connections to push consumers or pull suppliers.

VisiNotify restores persistent channel, admin and proxy as well as their current settings and IDs (ChannelID, AdminID and ProxyID). VisiNotify also reestablish transport connections. VisiNotify also supports an extended feature that automatically put a proxy on suspended state if the connection to the connected push consumer or pull supplier is broken. This is a better scenario than trying to reestablish the transport connection in a loop.

Self-adaptive asynchronous flow control

In OMG Notification Service 1.0, the channel should pull for event messages from a pull supplier when there is at least one consumer in the event stream. In OMG Notification Service 1.3, OMG requires the proxy to pull regardless whether any consumers are connected to the channel. The argument made from OMG is, that this pulling of events will shield the supplier from its consumers by preventing the supplier to know whether any consumers are present.

These two scenarios could lead to system and network resource waste on the unnecessary and tight pulling. However, with self-adaptive asynchronous flow control, the proxy pull consumer will only pull when returned events can be handed over to at least one consumer in the downstream. This implementation requires that each logic channel is assigned a voting slot. An upstream proxy pull consumer only pulls when the count in its voting slot is non zero. Each downstream proxy supplier, either push or pull, has one vote to its logic channel's voting slot. It votes to pull when the number of events in its queue is lower than the low watermark. And it withdraws its vote when pending events in its queue is more than the high watermark. This avoids the upstream proxy pull consumer to pull events back only to be discard or reject by downstream proxy consumers immediately. By setting the high and lower watermark, application can also get OMG Notification Service 1.0 or 1.3 behaviors.

QoS and filter support

VisiNotify supports OMG QoS and VisiNotify extensions. Also, VisiNotify provides a highly optimized and OMG compliant filter support for structured, sequence, and typed channels. See the Setting the Quality of Service and Filters chapter.

Thread on demand

Internally, channels and active proxies (proxy pull consumer and proxy push supplier) all require threads. However, threads are not assigned to them as dedicated servant. They are recycled when other objects above them (hierarchically) are in idle state. VisiNotify provides threads dynamically.

2

Developing supplier and consumer applications

This chapter discusses how you can develop supplier and consumer application using OMG Notification Service. The following topics are covered:

- Using predefined Event/Notification Services
- Using Typed Event/Notification Service
- Developing RMI/EJB applications with VisiNotify
- VisiBroker Event Buffering/Batch

Using predefined Event/Notification Services

The OMG Notification Service specifies three kinds of predefined channels, namely **Untyped** event channel, **Structured** event channel, and **Sequence** event channel. The advantage of predefined channel is that they are easy for user level implementations. Thereby, almost all notification service products on the market support predefined channels. The disadvantage of predefined channels are:

- They are slower than user defined typed channels.
- They usually have larger event size.
- They require more type unsafe dynamic manual code to pack and unpack user data into and from events.
- They do not have a formal, unified, widely adopted event description language.

For these reasons, the predefined untyped, structured and sequence channels are not good choice for new CORBA applications. However, they are support by VisiNotify for OMG compliance as well as for legacy applications. New applications should consider using the OMG Typed Notification Service. See the section Using Typed Event/ Notification Service for detailed information.

Developing push consumer applications

A push consumer is essentially a CORBA callback server application. It provides an push consumer object implementation. The push consumer object implementation

supports a predefined (untyped, structured or sequence) push consumer interface. The consumer application connects this consumer object to a channel to receive events.

Developing a push consumer application involve two tasks:

- (Optional) Implement a push supplier server object which support a pre-defined (untyped, structured or sequence) push supplier interface This involves:
  - Implementing the supplier servant.
  - Activating the servant on POA.
  - Activating the POA manager.
- Connect the consumer object to a channel. This involves:
  - Getting the channel reference.
  - Getting consumer admin from channel.
  - Obtaining proxy push supplier.
  - Connecting the consumer object to the proxy push supplier.

To illustrate the development of the push consumer application, the structured push consumer is used.

C++ Note

The push consumer example is located in examples/vbroker/notify/basic_cpp/structPushConsumer.C.

```
// 1. Implementing the push consumer servant
   class StructuredPushConsumerImpl : public
POA_CosNotifyComm::StructuredPushConsumer,
                                      public virtual
PortableServer::RefCountServantBase
   {
      ...
     public:
      ...
      void push_structured_event(const CosNotification::StructuredEvent& event)
        { ... }
      ...
   };

   // The consumer server
   int main(int argc, char** argv)
   {
       // get orb and POA ...
       ...
       // allocate a push consumer servant
       StructuredPushConsumerImpl* servant = new StructuredPUshConsumerImpl;

       // 2. activate the consumer servant on a POA
       poa->activate_object(servant);

       // 3. activate the POA
       poa->the_POAManager()->activate();
       ...
       // 4. somehow, we get the channel from somewhere
       CosNotifyChannelAdmin::EventChannel_var channel = ...;


       // 5. somehow, we decide to use the default admin
       CosNotifyChannelAdmin::ConsumerAdmin_var admin
            = channel->default_consumer_admin();
```

```
    // 6. obtain a proxy push supplier from the admin
    CosNotifyChannelAdmin::ProxyID pxy_id;
    CosNotifyChannelAdmin::ProxySupplier_var proxy
         = admin->obtain_notification_push_supplier(
                 CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

    CosNotifyChannelAdmin::StructuredProxyPushSupplier_var supplier
         = CosNotifyChannelAdmin::StructuredProxyPushSupplier::
           _narrow(proxy);

    // 7. get consumer object reference and connect it to the proxy
    CORBA::Object_var obj = poa->servant_to_reference(servant);
    CosNotifyComm::StructuredPushConsumer_var consumer
         = CosNotifyComm::StructuredPushConsumer::_narrow(obj);
    supplier->connect_structured_push_consumer(consumer);

    // working loop
    orb->run();
}
```

Java Note

The push consumer example is located in examples/vbroker/notify/basic_java/StructPushConsumer.java.

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPushConsumer extends StructuredPushConsumerPOA
{
   ...
  // 1. implement the push consumer servant
        public void push_structured_event(StructuredEvent event) { ... }
   ...
  public static int main(String[] args) {
    // get orb and POA ...
    ...
    // allocate a push consumer servant
    StructuredPushConsumer servant = new StructuredPUshConsumer();

    // 2. Activate the consumer servant on a POA
    poa.activate_object(servant);

    // 3. Activate the POA
    poa.the_POAManager().activate();
    ...
    // 4. Somehow, we get the channel from somewhere
    EventChannel channel = ...;

    // 5. Somehow, we decide to use the default admin
    ConsumerAdmin admin = channel.default_consumer_admin();

    // 6. Obtain a proxy push supplier from the admin
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxySupplier proxy = admin.obtain_notification_push_supplier(
                ClientType.STRUCTURED_EVENT, pxy_id);


    StructuredProxyPushSupplier supplier
```

```
            = StructuredProxyPushSupplierHelper.narrow(proxy);

        // 7. get consumer object reference and connect it to the proxy
        org.omg.CORBA::Object obj = poa.servant_to_reference(servant);
        StructuredPushConsumer consumer
            = StructuredPushConsumerHelper.narrow(obj);
        supplier.connect_structured_push_consumer(consumer);

        // working loop
        orb.run();
      }
    }
```

Develop pull consumer applications

A pull consumer is essentially a CORBA client. It obtains a proxy object in the channel and actively send request to the proxy to retrieve buffered events.

Developing a pull consumer application involve two tasks:

- (Optional) Implement a pull consumer server object which support a predefined (untyped, structured or sequence) pull consumer interface. This involves:
  - Implementing the consumer servant.
  - Activating the servant on POA.
  - Activating the POA manager.
- Get a proxy consumer reference and retrieve events from it. This involves:
  - Getting the channel reference.
  - Getting consumer admin from channel.
  - Obtaining proxy pull supplier.
  - Connecting the consumer object (or null) to the proxy pull supplier.
  - Actively pull event from the proxy pull supplier.

To illustrate the development of the pull consumer application, the structured pull consumer is used.

C++ Note

The pull consumer example is located in examples/vbroker/notify/basic_cpp/structPullConsumer.C.

```
// The consumer client
   int main(int argc, char** argv)
   {
       // get orb ...
       ...
       // 1. somehow, we get the channel from somewhere
       CosNotifyChannelAdmin::EventChannel_var channel = ...;

       // 2. somehow, we decide to use the default admin
       CosNotifyChannelAdmin::ConsumerAdmin_var admin
            = channel->default_consumer_admin();

       // 3. obtain a proxy pull supplier from the admin
       CosNotifyChannelAdmin::ProxyID pxy_id;
       CosNotifyChannelAdmin::ProxySupplier_var proxy
            = admin->obtain_notification_pull_supplier(
                    CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);
```

```
    CosNotifyChannelAdmin::StructuredProxyPullSupplier_var consumer
         = CosNotifyChannelAdmin::StructuredProxyPullConsumer::
           _narrow(proxy);

    // 4. connect to the proxy
    consumer->connect_structured_pull_consumer(NULL);

    // 5. Pull events from the proxy pull supplier
    for(int i=0;i<100;i++) {
       CosNotification::StructuredEvent_var event;
       event = supplier->pull_structured_event();
       ...
    }

    // 6. gracefully cleanup
    supplier->disconnect_structured_pull_supplier();
}
```

Java Note

The pull consumer example is located in `examples/vbroker/notify/basic_java/StructPullConsumer.java`.

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPullConsumer
{
   ...
 public static int main(String[] args) {
    // get orb ...
    ...
    // 1. Somehow, we get the channel from somewhere
    EventChannel channel = ...;

    // 2. Somehow, we decide to use the default admin
    ConsumerAdmin admin = channel.default_consumer_admin();

    // 3. Obtain a proxy pull supplier from the admin
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxySupplier proxy = admin.obtain_notification_pull_supplier(
                ClientType.STRUCTURED_EVENT, pxy_id);

    StructuredProxyPullSupplier supplier
         = StructuredProxyPullSupplierHelper.narrow(proxy);

    // 4. connect to the proxy
    supplier.connect_structured_pull_consumer(null);

    // 5. Pull events from the proxy pull supplier
    for(int i=0;i<100;i++) {
       StructuredEvent event = supplier.pull_structured_event();
       ...
    }

    // 6. gracefully cleanup
    supplier.disconnect_structured_pull_supplier();
 }
}
```

Developing push supplier applications

A push supplier application is a CORBA client. It actively invokes request on a proxy consumer object to send events to the channel.

Developing a push supplier application involve two tasks:

- (Optional) Implement a push supplier server object which support a predefined (untyped, structured or sequence) push supplier interface. This involves:
  - Implementing the supplier servant.
  - Activating the servant on POA.
  - Activating the POA manager.
- Get a proxy supplier reference and send events to it. This involves:
  - Getting the channel reference.
  - Getting supplier admin from channel.
  - Obtaining proxy push consumer.
  - Connecting the supplier object (or null) to the proxy push consumer.
  - Actively push events to the proxy push consumer.

To illustrate the development of the push supplier application, the structured push supplier is used.

C++ Note

The push supplier example is located in examples/vbroker/notify/basic_cpp/structPushSupplier.C.

```
// The push supplier client
   int main(int argc, char** argv)
   {
       // get orb ...
       ...
       // 1. somehow, we get the channel from somewhere
       CosNotifyChannelAdmin::EventChannel_var channel = ...;

       // 2. somehow, we decide to use the default admin
       CosNotifyChannelAdmin::SupplierAdmin_var admin
            = channel->default_supplier_admin();

       // 3. obtain a proxy push consumer from the admin
       CosNotifyChannelAdmin::ProxyID pxy_id;
       CosNotifyChannelAdmin::ProxyConsumer_var proxy
            = admin->obtain_notification_push_consumer(
                    CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

       CosNotifyChannelAdmin::StructuredProxyPushConsumer_var Consumer
            = CosNotifyChannelAdmin::StructuredProxyPushConsumer::
                  _narrow(proxy);

       // 4. connect to the proxy
consumer->connect_structured_push_supplier(NULL);

       // 5. Push events to the proxy push consumer
       for(int i=0;i<100;i++) {
          CosNotification::StructuredEvent_var event = ...;
          Consumer->push_structured_event(event);
       }
```

```
        // 6. Gracefully cleanup
        consumer->disconnect_structured_push_consumer();
    }
```

Java Note

The push supplier example is located in examples/vbroker/notify/basic_java/StructPushSupplier.java.

```
import org.omg.CosNotifyComm.*;
import org.omg.CosNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class StructuredPushSupplier
{
   ...
 public static int main(String[] args) {
    // get orb ...
    ...
    // 1. Somehow, we get the channel from somewhere
    EventChannel channel = ...;

    // 2. Somehow, we decide to use the default admin
    ConsumerAdmin admin = channel.default_supplier_admin();

    // 3. Obtain a proxy consumer from the admin
    ProxyIDHolder pxy_id = new ProxyIDHolder();
    ProxyConsumer proxy = admin.obtain_notification_push_consumer (
                ClientType.STRUCTURED_EVENT, pxy_id);

    StructuredProxyPushConsumer consumer
         = StructuredProxyPushConsumerHelper.narrow(proxy);

    // 4. Connect to the proxy
    consumer.connect_structured_push_supplier(null);

    // 5. Push events to the proxy push consumer
    for(int i=0;i<100;i++) {
       StructuredEvent event = ...;
       Consumer.push_structured_event(event);
    }

    // 6. Gracefully cleanup
    consumer.disconnect_structured_push_consumer();
  }
}
```

Developing pull supplier applications

A pull supplier application is a CORBA callback server. It provides an pull supplier object implementation. The pull supplier object implementation supports a predefined (untyped, structured or sequence) pull supplier interface. The supplier application needs to connect this supplier object to a channel to supply events.

Developing a pull supplier application involve two tasks:

- Implement a normal pull supplier server object which support a predefined (untyped, structured or sequence) pull supplier interface. This involves:
  - Implementing the supplier servant.
  - Activating the servant on POA.

  - Activating the POA manager.
- Connect the supplier object to a channel. This involves:
  - Getting the channel reference.
  - Getting supplier admin from channel.
  - Obtaining proxy pull consumer.
  - Connecting the pull supplier object to the proxy pull consumer.

To illustrate the development of the pull supplier application, the structured pull supplier is used.

C++ Note

The pull supplier example is located in `examples/vbroker/notify/basic_cpp/structPullSupplier.C`.

```
// 1. Implement the pull supplier servant
   class StructuredPullSupplierImpl : public
POA_CosNotifyComm::StructuredPullSupplier,
                                    public virtual
PortableServer::RefCountServantBase
   {
      ...
     public:
      ...
       CosNotification::StructuredEvent* pull_structured_event() { ... }
       CosNotification::StructuredEvent* try_pull_structured_event
           ( CORBA::Boolean& has_event) { ... }
      ...
   };

   // The supplier server
   int main(int argc, char** argv)
   {
       // get orb and POA ...
       ...
       // allocate a pull supplier servant
       StructuredPullSupplierImpl* servant = new StructuredPullSupplierImpl;

       // 2. activate the consumer servant on a POA
       poa->activate_object(servant);

       // 3. activate the POA
       poa->the_POAManager()->activate();
       ...
       // 4. somehow, we get the channel from somewhere
       CosNotifyChannelAdmin::EventChannel_var channel = ...;

       // 5. somehow, we decide to use the default admin
       CosNotifyChannelAdmin::SupplierAdmin_var admin
            = channel->default_supplier_admin();

       // 6. obtain a proxy pull consumer from the admin
       CosNotifyChannelAdmin::ProxyID pxy_id;
       CosNotifyChannelAdmin::ProxyConsumer_var proxy
            = admin->obtain_notification_pull_consumer(
                    CosNotifyChannelAdmin::STRUCTURED_EVENT, pxy_id);

       CosNotifyChannelAdmin::StructuredProxyPullConsumer_var consumer
            = CosNotifyChannelAdmin::StructuredProxyPullConsumer::
```

```
                    _narrow(proxy);


        // 7. get supplier object reference and connect it to the proxy
        CORBA::Object_var obj = poa->servant_to_reference(servant);
        CosNotifyComm::StructuredPullSupplier_var supplier
             = CosNotifyComm::StructuredPullSupplier::_narrow(obj);
        consumer->connect_structured_pull_supplier(supplier);

        // working loop
        orb->run();
    }
```

Java Note

The pull supplier example is located in `examples/vbroker/notify/basic_java/StructPullSupplier.java`.

```
    import org.omg.CosNotifyComm.*;
    import org.omg.CosNotifyChannelAdmin,*;
    import org.omg.CosNotification.*;

    public class StructuredPullSupplier extends StructuredPullSupplierPOA
    {
       ...
      // 1. implement the push consumer servant
      public StructuredEvent pull_structured_event() { ... }
      public StructuredEvent try_pull_structured_event
              (org.omg.CORBA.BooleanHolder has_event) {...}
       ...
      public static int main(String[] args) {
        // get orb and POA ...
        ...
        // allocate a pull supplier servant
        StructuredPullSupplier servant = new StructuredPullSupplier();

        // 2. Activate the supplier servant on a POA
        poa.activate_object(servant);

        // 3. Activate the POA
        poa.the_POAManager()Activate();
        ...
        // 4. Somehow, we get the channel from somewhere
        EventChannel channel = ...;

        // 5. Somehow, we decide to use the default admin
        ConsumerAdmin admin = channel.default_supplier_admin();

        // 6. Obtain a proxy pull consumer from the admin
        ProxyIDHolder pxy_id = new ProxyIDHolder();
        ProxyConsumer proxy = admin.obtain_notification_pull_consumer(
                    ClientType.STRUCTURED_EVENT, pxy_id);

        StructuredProxyPullConsumer consumer
             = StructuredProxyPullConsumerHelper.narrow(proxy);

        // 7. Get supplier object reference and connect it to the proxy
        org.omg.CORBA::Object obj = poa.servant_to_reference(servant);
        StructuredPullSupplier supplier
            = StructuredPullSupplierHelper.narrow(obj);
        consumer.connect_structured_pull_supplier(supplier);
```

```
        // working loop
        orb.run();
    }
}
```

Using Typed Event/Notification Service

The predefined events (untyped, structured, sequence) in OMG Event/Notification Service present a message-oriented approach. The disadvantages of this approach are:

- They are slower than user defined typed channels.
- They usually have larger event size.
- They require more type unsafe dynamic manual code to pack and unpack user data into/from events.
- They do not have a formal, unified, widely adopted event description language.

Therefore, in developing new applications, it is recommended to use a user defined typed event and the OMG Typed Event/Notification Service. By using the OMG Typed Event/Notification Service, event interfaces are not predefined by OMG but rather by user applications using OMG IDL language. Using this approach will results in the following:

- Application event throughput can be significantly higher.
- Event size can be substantially smaller.
- Event pack and unpack operation utilize type safe IDL generated static stub/skeleton code.
- Events are formally defined by IDL.

There are minor short falls in using the OMG Typed Event/Notification service. They include:

- Connecting to Typed Event/Notification Service is slightly complicated than connecting to (predefined) Event/Notification Service. Applications need to provide an additional handler implementation or/and perform additional connection operations (for example, get_typed_consumer()). However, the tradeoff to get mentioned advantages is worthwhile.
- Directly using Event/Notification service to do typed pulling is not appropriately defined by OMG. The OMG solution requires substantial work.

VisiBroker Publish/Subscribe Adapter (PSA) architecture resolves these two issue. See Publish/Subscribe Adapter (PSA) for additional information. The PSA simplifies and unifies the connection procedure to notification and typed notification services. It also presents an elegant solution for typed pulling.

Note

This chapter only discusses how to develop typed push applications directly using OMG Typed Notification Service. Typed pulling and PSA will be discussed in PSA chapter.

As the user defined event type, the following IDL interface definition is used throughout the examples in this chapter:

```
// TMN.idl: typed event definition

// user defined pragma
# pragma prefix "example.borland.com"

// user defined module
```

```
module TMN {




      // user defined event interface
      interface TypedEvent {
            void attributeValueChange(...);
            void qosAlarm(...);
            ...
      };
};
```

Developing type push consumer applications

A typed push consumer is essentially a CORBA callback server application. It provides an user defined typed consumer object implementation. The typed push consumer object implementation supports the user defined IDL interface. The consumer application connects this consumer object to a typed channel to receive typed events.

Developing a typed push consumer application involves two tasks:

- Implement a normal consumer server object which support a user defined IDL interface. This involves:
  - Implementing the user defined typed consumer servant, such as the <I> interface servant.
  - Implementing a handler servant. This handler servant support the CosTypedNotifyComm::TypedPushConsumer interface and its get_typed_consumer() operation, which returns a reference of the user defined typed consumer object (for example, returns the <I> interface).
  - Activating the user defined typed servant on a POA and get its reference.
  - Activating this handler object and pass it to the user defined typed consumer object reference (such as the <I> interface).
  - Activating the POA manager.
- Connect the consumer object to a channel. This involves:
  - Getting the typed channel reference.
  - Getting typed consumer admin from channel.
  - Obtaining typed proxy push supplier.
  - Connecting the handler object to the typed proxy push supplier.

The following example compares the procedure of using predefined event interface. Notice that using typed event requires an additional implementation on a push consumer application.

C++ Note

The typed push consumer example is located in `examples/vbroker/notify/basic_cpp/typedPushConsumer.C`.

```
// 1. Implement the user defined typed consumer servant
class TMNTypedEventImpl : public POA_TMN::TypedEvent,
                          public virtual PortableServer::RefCountServantBase
{
   ...
  public:
   ...
```

```
   void attributeValueChange (...) { ... }
   void qosAlarm(...) { ... }
   ...
};


// 2. Implement the handler servant
class HandlerImpl : public POA_CosTypedNotifyComm::TypedPushConsumer,
                    public virtual PortableServer::RefCountServantBase
{
    CORBA::Object_var   _the_typed_consumer; // the <I> interface

  public:
    HandlerImpl(CORBA::Object_ptr ref)
      : _the_typed_consumer(CORBA::Object::_duplicate(ref)) {}

    CORBA::Object_ptr get_typed_consumer() {
              // return the <I> interface
              return CORBA::Object::_duplicate(_the_typed_consumer); }
    ...
};

// The typed consumer server
int main(int argc, char** argv)
{
    // get orb and POA ...
    ...
    // allocate a push consumer servant
    TMNTypedEventImpl* servant = new TMNTypedEventImpl;

    // 3. activate the typed consumer on a POA
    poa->activate_object(servant);

    // 4. Get typed consumer reference
    CORBA::Object_var obj = poa->servant_to_reference(servant);

    // 5. allocate a handler servant and pass it the typed
          consumer reference
    HandlerImpl* handler = new HandlerImpl(obj);

    // 6. Activate the handler object on a POA
    poa->activate_object(handler);

    // 7. Activate the POA(s)
    poa->the_POAManager()->activate();
    ...
    // 8. somehow, we get a typed channel from somewhere
    CosTypedNotifyChannelAdmin::TypedEventChannel_var channel = ...;

    // 9. somehow, we decide to use the default admin
    CosTypedNotifyChannelAdmin::TypedConsumerAdmin_var admin
         = channel->default_consumer_admin();

    // 10. obtain a proxy push supplier from the admin using the event
    //    repository id "IDL:example.borland.com/TMN/TypedEvent:1.0"
    //    as the key.
    CosNotifyChannelAdmin::ProxyID pxy_id;
    CosTypedNotifyChannelAdmin::ProxySupplier_var proxy
         = admin->obtain_typed_notification_push_supplier(
             "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id);
```

```
    // 11. get handler object reference and connect it to the proxy
    CORBA::Object_var ref = poa->servant_to_reference(handler);
    CosTypedNotifyComm::TypedPushConsumer_var consumer
         = CosTypedNotifyComm::TypedPushConsumer::_narrow(ref);
    proxy->connect_typed_push_consumer(consumer);

    // working loop
    orb->run();
}
```

Java Note

The typed push consumer example is located in examples/vbroker/notify/basic_java/TypedPushConsumerImpl.java.

```
// 1. Implement the user defined typed consumer servant
class TMNTypedEventImpl extends TMN.TypedEventPOA {
   ...
   public void attributeValueChange (...) { ... }
   public void qosAlarm(...) { ... }
   ...
}

// 2. Implement the handler servant
public class TypedPushConsumerImpl
         extends org.omg.CosTypedNotifyComm.TypedPushConsumer {
    org.omg.CORBA.Object _the_typed_consumer = null; // the <I> interface

    TypedPushConsumerImpl(org.omg.CORBA.Object ref) {
       _the_typed_consumer = ref;
    }
    org.omg.CORBA.Object get_typed_consumer() {
        // return the <I> interface
        return _the_typed_consumer; }
    ...
    public static void main(int argc, char** argv) {
       // get orb and POA ...
       ...
       // allocate a push consumer servant
       TMNTypedEventImpl servant = new TMNTypedEventImpl();

       // 3. activate the typed consumer on a POA
       poa.activate_object(servant);

       // 4. Get typed consumer reference
       org.omg.CORBA.Object obj = poa.servant_to_reference(servant);

       // 5. allocate a handler servant and pass it the typed
             consumer reference
       TypedPushConsumerImpl handler = new TypedPushConsumerImpl(obj);

       // 6. Activate the handler object on a POA
       poa.activate_object(handler);

       // 7. Activate the POA(s)
```

```
        poa.the_POAManager()Activate();
        ...
        // 8. somehow, we get a typed channel from somewhere
        org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel channel = ...;

        // 9. somehow, we decide to use the default admin
        org.omg.CosTypedNotifyChannelAdmin.TypedConsumerAdmin admin
            = channel.default_consumer_admin();
        // 10. Obtain a proxy push supplier from the admin using the event
        //    repository id "IDL:example.borland.com/TMN/TypedEvent:1.0"
        //    as the key.
        org.omg.CosNotifyChannelAdmin.ProxyIDHolder pxy_id_holder
          = new org.omg.CosNotifyChannelAdmin.ProxyIDHolder();
        org.omg.CosTypedNotifyChannelAdmin.ProxySupplier proxy
          = admin.obtain_typed_notification_push_supplier(
              "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id_holder);

        // 11. Get handler object reference and connect it to the proxy
        org.omg.CORBA.Object ref = poa.servant_to_reference(handler);
        org.omg.CosTypedNotifyComm.TypedPushConsumer consumer
          = org.omg.CosTypedNotifyComm.TypedPushConsumerHelper.narrow(ref);
        proxy.connect_typed_push_consumer(consumer);

        // working loop
        orb.run();
    }
}
```

Developing typed push supplier applications

A typed push supplier application is a CORBA client. It actively invokes request on a typed consumer proxy object to send typed events to the channel.

Developing a typed push supplier application involves two tasks:

- (Optional) Implement a typed push supplier server object. This involves:
  - Implementing the supplier servant.
  - Activating the servant on POA
  - Activating the POA manager.
- Get the proxy supplier reference and send event to it.. This involves:
  - Getting the typed channel reference.
  - Getting supplier admin from the typed channel.
  - Obtaining typed proxy push consumer.
  - Calling get_typed_consumer() on the typed proxy push consumer to get the <I> interface reference.
  - Actively push events to the <I> interface reference.

The following example compares the procedure of using predefined event interface. Notice that using typed event requires an additional procedure, such as get_typed_consumer().

C++ Note

The typed push supplier example is located in `examples/vbroker/notify/basic_cpp/typedPushSupplier.C`.

```
// The typed push supplier client
int main(int argc, char** argv)
```

```
{
    // get orb ...
    ...
    // 1. Somehow, we get the typed channel from somewhere
    CosTypedNotifyChannelAdmin::TypedEventChannel_var channel = ...;

     // 2. Somehow, we decide to use the default admin
    CosTypedNotifyChannelAdmin::TypedSupplierAdmin_var admin
        = channel->default_supplier_admin();
    // 3. obtain a typed proxy push consumer from the admin using the event
    //    repository id "IDL:example.borland.com/TMN/TypedEvent:1.0"
    //    as the key.
    CosTypedNotifyChannelAdmin::ProxyID pxy_id;
    CosTypedNotifyChannelAdmin::TypedProxyPushConsumer_var proxy
         = admin->obtain_typed_notification_push_consumer(
                "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id);

     // 4. connect to the proxy
    proxy->connect_typed_push_supplier(NULL);

     // 5. Get the <I> interface
    CORBA::Object_var obj = proxy->get_typed_consumer();
    TMN::TypedEvent_var consumer = TMN::TypedEvent::_narrow(obj);

     // 6. Push events to the <I> interface
    for(int i=0;i<100;i++) {
       consumer->attributeValueChange(...);
       consumer->qosAlarm(...);
    }

     // 7. Flush buffered events
    consumer->_non_existent();

     // 8. Gracefully cleanup
    proxy->disconnect_typed_push_consumer();
}
```

Java Note

The typed push consumer example is located in `examples/vbroker/notify/basic_java/TypedPushSupplier.java`.

```
import org.omg.CosTypedNotifyComm.*;
import org.omg.CosTypedNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class TypedPushSupplierImpl
{
   ...
   // The typed push supplier client
   public static void main(String[] args) {
      // get orb ...
      ...
       // 1. Somehow, we get the typed channel from somewhere
      org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel_var channel
          = ...;

       // 2. Somehow, we decide to use the default admin
      org.omg.CosTypedNotifyChannelAdmin.TypedSupplierAdmin admin
          = channel.default_supplier_admin();
```

```
            // 3. obtain a typed proxy push consumer from the admin
                  using the event
            //    repository id "IDL:example.borland.com/TMN/TypedEvent:1.0"
            //    as the key.
            Org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder pxy_id
                              = new
org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder();
            CosTypedNotifyChannelAdmin::TypedProxyPushConsumer_var proxy
               = admin.obtain_typed_notification_push_consumer(
                        "IDL:example.borland.com/TMN/TypedEvent:1.0", pxy_id);

            // 4. connect to the proxy
            proxy.connect_typed_push_supplier(null);

            // 5. Get the <I> interface
            org.omg.CORBA.Object obj = proxy.get_typed_consumer();
            TMN.TypedEvent consumer = TMN.TypedEventHelper.narrow(obj);

            // 6. Push events to the <I> interface
            for(int i=0;i<100;i++) {
               consumer.attributeValueChange(...);
               consumer.qosAlarm(...);
            }

            // 7. Flush buffered events
            consumer._non_existent();

            // 8. Gracefully cleanup
            proxy.disconnect_typed_push_consumer();
        }
    }
```

Developing RMI/EJB applications with VisiNotify

With the introduction of J2EE 1.3, RMI-over-IIOP has been standardized in the J2EE implementations. Therefore, the interoperation and interconnection between CORBA and J2EE environments have become seamlessJ2EE is basically a framework for client/server applications. However, J2EE technology does not provide adequate support for publish/subscribe applications. The only solutions defined in J2EE are Java Messaging Service (JMS) and Message Driver Bean (MDB). JMS is purely a message-oriented service that is mainly used for integrating or interconnecting with legacy message middleware. MDB is simply defined following the use of JMS and allows legacy message middleware to send messages to an enterprise bean through JMS. In this regard, the JMS and MDB based solutions usually share the same disadvantages of legacy message oriented middlewares. They include:

- They are slower than user defined object-oriented typed channels.
- They have relatively larger event size.
- They require more nonstandard or type unsafe dynamic manual code to pack and unpack user data into/from events.
- They do not have a formal, unified, widely adopted event description language.

The OMG Typed Event/Notification Service resolves these issues. A typed notification can be used as a publish/subscribe middleware for RMI/EJB application. In addition, VisiNotify offers support for direct connection between OMG Structured channel and RMI/EJB. Besides, VisiNotify provides direct support of CORBA valuetypes (either in standard marshalling or in customer marshalling) as well as Java serializeable objects. With these standard facilities from OMG Typed Event/Notification Service, J2EE 1.3, and VisiNotify extensions, event driven RMI/EJB applications can be developed as

normal object oriented applications rather than mapping OMG Notification Service as a JMS provider and then using JMS/MDB . The advantages of this approach are:

- Significant performance improvement.
- Smaller event size.
- Static type safe RMI stubs and skeleton perform message pack/unpack.
- Event are described by user defined Java RMI interface.

This section describes how OMG Typed Event/Notification Service VisiNotify is used in the RMI/EJB environments.

This user defined Java RMI remote interface is used as either an RMI server interface or an EJB consumer bean remote interface throughout the examples in this section. .

```
package TMN;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Notification extends Remote {
        void attributeValueChange(...) throws RemoteException;
        void qosAlarm(...) throws RemoteException;
        ...
}
```

Developing a RMI typed consumer

A RMI typed push consumer is essentially a RMI callback application connected to the OMG Typed Notification service. The typed push consumer RMI object implements user defined RMI interface. A RMI typed consumer is very similar to a CORBA typed consumer with slight differences. They include:

- The RMI object does not need to be explicitly activated on POA.
- The application needs to get RMI object's CORBA object reference as the <I> interface (see step 4 in example below).

This code example shows a RMI typed push consumer:

```
 // 1. Implement the user defined typed consumer RMI object
class RMINotifyImpl
      extends PortableRemoteObject
      implements TMN.Notification {
   ...
   public void attributeValueChange (...) { ... }
   public void qosAlarm(...) { ... }
   ...
}

 // 2. Implement the handler servant
public class TypedPushConsumerImpl
         extends org.omg.CosTypedNotifyComm.TypedPushConsumer {
    org.omg.CORBA.Object _the_typed_consumer = null; // the <I> interface

    TypedPushConsumerImpl(org.omg.CORBA.Object ref) {
       _the_typed_consumer = ref;
    }
    org.omg.CORBA.Object get_typed_consumer() {
        // return the <I> interface
        return _the_typed_consumer; }
    ...
    public static void main(int argc, char** argv) {
       // get orb and POA ...
```

```
        ...
         // 3. allocate a RMI consumer object
        RMINotifyImpl consumer = new RMINotifyImpl();

         // 4. get the CORBA object reference of the RMI consumer
        org.omg.CORBA.Object corba_obj
                = javax.rmi.CORBA.Util.getTie(consumer).thisObject();
         // 5. allocate a handler servant and pass it the typed consumer
                                                                    reference
        TypedPushConsumerImpl handler = new TypedPushConsumerImpl(corba_obj);

         // 6. Activate the handler object on a POA
        poa.activate_object(handler);

         // 6. Activate the POA(s)
        poa.the_POAManager()Activate();
        ...
         // 7. somehow, we get a typed channel from somewhere
        org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel channel = ...;

         // 8. somehow, we decide to use the default admin
        org.omg.CosTypedNotifyChannelAdmin.TypedConsumerAdmin admin
              = channel.default_consumer_admin();

         // 9. obtain a proxy push supplier from the admin
        org.omg.CosNotifyChannelAdmin.ProxyIDHolder pxy_id_holder
              = new org.omg.CosNotifyChannelAdmin.ProxyIDHolder();
        org.omg.CosTypedNotifyChannelAdmin.ProxySupplier proxy
              = admin.obtain_typed_notification_push_supplier
                 ( "RMI.Test" , pxy_id_holder);

         // 10. Get handler object reference and connect it to the proxy
        org.omg.CORBA.Object ref = poa.servant_to_reference(handler);
        org.omg.CosTypedNotifyComm.TypedPushConsumer consumer
          = org.omg.CosTypedNotifyComm.TypedPushConsumerHelper.narrow(ref);
        proxy.connect_typed_push_consumer(consumer);

        // working loop
        orb.run();
    }
}
```

Developing a RMI typed supplier

A RMI typed supplier is very similar to its CORBA counterpart except for the <I> reference, which is returned from get_typed_consumer(), should be narrowed into the correspondent RMI stub (see step 6 in example below).

This code example shows a RMI typed push supplier:

```
import org.omg.CosTypedNotifyComm.*;
import org.omg.CosTypedNotifyChannelAdmin,*;
import org.omg.CosNotification.*;

public class TypedPushSupplierImpl
{
   ...
   // The typed push supplier client
   public static void main(String[] args) {
      // get orb ...
```

```
        ...
         // 1. Somehow, we get the typed channel from somewhere
        org.omg.CosTypedNotifyChannelAdmin.TypedEventChannel_var channel
               = ...;



         // 2. Somehow, we decide to use the default admin
        org.omg.CosTypedNotifyChannelAdmin.TypedSupplierAdmin admin
               = channel.default_supplier_admin();

         // 3. obtain a typed proxy push consumer from the admin
        Org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder pxy_id
                          = new
org.omg.CosTypedNotifyChannelAdmin.ProxyIDHolder();
        CosTypedNotifyChannelAdmin::TypedProxyPushConsumer_var proxy
           = admin.obtain_typed_notification_push_consumer(
                   "RMI.Test, pxy_id);

         // 4. connect to the proxy
        proxy.connect_typed_push_supplier(null);

         // 5. Get the <I> interface
        org.omg.CORBA.Object obj = proxy.get_typed_consumer();

        // 6. Narrowing the CORBA object reference into RMI stub.
        TMN.Notification consumer = (TMN.Notification)PortableRemoteObject.
                                       narrow(obj, TMN.Notification.class);

         // 7. Push events to the <I> RMI stub
        for(int i=0;i<100;i++) {
           consumer.attributeValueChange(...);
           consumer.qosAlarm(...);
        }

         // 8. Flush buffered events
        com.inprise.vbroker.orb.BufferedEvents.flush();

         // 9. Gracefully cleanup
        proxy.disconnect_typed_push_consumer();
     }
  }
```

Developing an EJB bean as an Typed Notification consumer

An EJB typed event bean can be any type of bean (session or entity, stateless or stateful), except for an MDB. The EJB typed event bean implements event operations as declared in the given associated user defined remote interface.

This code example shows an EJB bean as a push consumer of user defined TMN.Notification remote interface:

```
 import javax.ejb.*;

// 1. The bean implementation
public class TMNNotifyBean implements SessionBean {
   ...
   // implement operations declared in bean's remote interface
   public void attributeValueChange (...) { ... }
   public void qosAlarm(...) { ... }
```

```
   ...
}
```

After building and deploying this typed EJB bean implementation, your application can:

- Get its home interfaces from JNDI name.
- Get its remote interface from the home interface.
- Connect its remote interface to an OMG Typed Notification channel.

In this release, a command line utility, *subtool*, is provided to subscribe an EJB bean as a typed RMI consumer by knowing its JNDI name. To connect a typed event EJB bean to an OMG Typed Notification channel, use subtool command:

```
% subtool [-channel <ior>|-admin <ior>] \
                -home <jndi_name> \
                -type typed \
                -key <proxy_key>
```

where,

- The `-channel` or `-admin` option specify the channel or consumer admin object as the subscribe point.
- The `-home` <jndi_name> tells subtool the JNDI name of the bean's Home interface.
- The `-type typed` option tells subtool to connect the bean's remote interface as typed consumer.
- The `-key <proxy_key>` option tells the subtool what should be the key parameter for obtain_typed_notification_push_supplier().

This example shows using subtool to subscribe a typed event bean to a OMG Typed Notification Channel:

```
% subtool  -channel  corbaloc::127.0.0.1:14100/default_channel \
         -home  stock_home  -type  typed  -key  Stock
```

Developing an EJB bean as a Structured Notification consumer

An EJB structured event bean can be any type of EJB bean (session or entity, stateless or stateful), except for an MDB. This EJB structured event bean can connect to an VisiNotify Structured Notification Service and receive structured event originated from non-RMI CORBA applications. An EJB structured event bean implements, among other mandatory operations, a void push_structured_event (org.omg.CosNotification.StructuredEvent) operation. This operation should not be overloaded in this bean and its remote interface.

Unlike typed event bean, support of structured event bean is a VisiNotify extension. VisiNotify does a special translation to convert a StructuredEvent structure that is sent into the channel from a CORBA Structured supplier application into a StructuredEvent valuebox when it detects the connected consumer is a structured event EJB bean.

This example shows an EJB bean as a structured push consumer:

```
import javax.ejb.*;

 // The bean implementation
public class MyStructuredNotifyBean implements SessionBean {
   ...
    public void push_structured_event(
                  org.omg.CosNotification.StructuredEvent event) { ... }
   ...
}
```

After building and deploying this structured event bean, connect its remote interface to the given VisiNotify's structured event channel. The remote interface of this bean should declare the push_structured_event() operation. This code example shows the connection as:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

 // The bean's remote interface
public interface MyStructuredInterface extends Remote {
    public void push_structured_event(
                org.omg.CosNotification.StructuredEvent event)  throws
RemoteException;
          ...
}
```

It is prohibited by ORB type system to directly connect this structure event bean's remote interface as a structured event consumer to OMG Notification Service structured channel. Therefore, to connect a structured event bean to a VisiNotify Structured Notification channel, use the subtool command:

```
% subtool [-channel <ior>|-admin <ior>] \
          -home <jndi_name> \
          -type struct
```

where,

- The `-channel` or `-admin` option specify the channel or consumer admin object as the subscribe point.
- The `-home` <jndi_name> tells subtool the JNDI name of the bean's Home interface.
- The `-type struct` option tells subtool to connect the bean's remote interface as structured consumer.

This example shows using subtool to subscribe a structured event bean to a VisiNotify Structured Notification Channel:

```
% subtool  -channel  corbaloc::127.0.0.1:14100/default_channel \
            -home  stock_home  -type struct
```

VisiBroker Event Buffering/Batch

Event buffering/batch is a mechanism implemented in VisiBroker 5.1 to optimize VisiNotify event throughput. By default, event are buffered in supplier-side stubs before flushed to VisiNotify as a larger batch message. Also, if VisiNotify detects that the consumer is working on top of VisiNotify 5.1, it will try to buffering/batch events together.

Disable supplier-side event buffering

Supplier applications can disable supplier-side event buffering by setting `vbroker.orb.supplier.eventBatch` to false. For example:

```
% typedPushSupplier ... -Dvbroker.orb.supplier.eventBatch=false
```

or

```
% vbj ... StructPushSupplier ... -Dvbroker.orb.supplier.eventBatch=false
```

Disable consumer-side event buffering

Consumer applications can also disable VisiNotify from sending events in batch by setting `vbroker.orb.consumer.eventBatch` to false. For example:

```
% typedPushConsumer ... -Dvbroker.orb.consumer.eventBatch=false
```

or

```
% vbj ... StructPushConsumerImpl ... -Dvbroker.orb.consumer.eventBatch=false
```

Flush buffered events in supplier application

The supplier-side VisiBroker runtime will flush an event when these conditions occur:

Event buffer is full: This is a per-stub level flush. The default size of this stub level event buffer is 32K. A given supplier application can use `vbroker.orb.supplier.eventBufferSize` to change this size between 8K and 64K. For example:

```
% typedPushSupplier ... -Dvbroker.orb.supplier.eventBufferSize=48000
```

Number of buffered events reaches the maximum batch size: This is a per-stub level flush. The default maximum number of events that a stub can hold in its buffer is 128. A supplier application can use `vbroker.orb.supplier.maxBatchSize` to change this size to any value less than 256. For example:

```
% vbj ... UntypedPushSupplier ... -Dvbroker.orb.supplier.eventBatchSize=32
```

Internal buffer flush timeout: This is a global flush. On timeout, all events buffered in all stubs will be flushed out. The default timeout interval is 2,000 milliseconds (2 seconds). A supplier application can use `vbroker.orb.supplier.eventBatchTimerInterval` to change this time between 100 millisecond (0.1 second) and 10,000 milliseconds (10 seconds). For example:

```
% typedPushSupplier ... -Dvbroker.orb.supplier.eventBatchTimerInterval=5000
```

Supplier invoked a non-bufferable operation on the stub: This is a per-stub level flush and includes:

- For untyped proxy consumer stub, only the push() operation is bufferable.
- For structured proxy consumer stub, only the push_structured_event() operation is bufferable.
- For sequence proxy consumer stub, only the push_structured_events() operation is bufferable.

Note

Therefore, invoking `disconnect_..._push_consumer()` operations or `_non_existent()` on the proxy stubs (above) will flush out all buffered events.

- For the <I> interface stub of a typed channel, all non-pseudo operations are bufferable.

Therefore, a supplier application can invoke `_non_existent()` operation on an <I> interface stub to flush its buffered events. Notice that the calling `disconnect_typed_push_consumer()` on a typed proxy consumer stub will not cause the buffer in a corresponding <I> stub to be flushed. The application should explicitly `call _non_existent()` on an <I> interface stub before calling `disconnect_typed_push_consumer()` on proxy stub.

Java application calling BufferedEvent.flush()

A Java supplier application can explicitly call `com.inprise.vbroker.orb.BufferedEvents.flush()` to flush. This is a global level event flush. It is to support VisiBroker RMI applications because there is no pseudo operation on a java.rmi.Remote interface, which can be used for event flush. Calling this static method will flush out all events in every stub.

Initial Reference of VisiNotify

By default, VisiNotify uses TCP port number 14100 unless `-Dvbroker.notify.listener.port=<port>` is used in the command line. Therefore, as specified by OMG Notification Service, the URL of the Channel factory and typed channel factory are:

```
corbaloc::<host>:14100/NotificationService
corbaloc::<host>:14100/TypedNotificationService
```

where, `<host>` is the domain name or dotted IP address of the VisiNotify host machine. The VisiNotify server also creates a default channel. The URL of this default channel is:

```
corbaloc::<host>:14100/default_channel
```

This URL can be registered to supplier or consumer application's ORB using these two OMG standardized scenarios:

1 –ORBInitRef ORB_init() command line option. Examples:

```
-ORBInitRef NotificationService=corbaloc::127.0.0.1:14100/NotificationService
```

or

```
-ORBInitRef TypedNotificationService=corbaloc::127.0.0.1:14100/
TypedNotificationService
```

2 ORB::register_initial_reference(). Examples:

```
    orb.register_initial_reference(
           "TypedNotificationService",
     orb.string_to_object("corbaloc::127.0.0.1:14100/
TypedNotificationService");
```

After registering them as an initial service, application can use `resolve_initial_reference()`.

3

Using the Publish Subscribe Adapter (PSA)

This chapter introduces VisiBroker Publish/Subscribe Adapter (PSA). The PSA is primarily a programming model and a component that works in conjunction with OMG Event/Notification Service. It is interoperable with applications that use low-level OMG Notification Service interfaces.

Introduction

As “one of the best client/server middleware products,” CORBA provides solid support for traditional client/server applications that are based on OMG object-oriented ORB architecture. However, there are some short falls with CORBA in respect to supporting publish/subscribe applications. For many enterprise business applications, the publish/subscribe communication model is as important as the client/server model. Direct support of the publish/subscribe communication model in the CORBA middleware infrastructure substantially reduces the development effort by allowing developers focus on implementing business logic rather than redesigning system solutions.

Notwithstanding, the ORB level support of the publish/subscribe communication model has been virtually omitted within OMG along with third-party ORB vendors. The publish/subscribe communication model is considered as a “second class subject” within the CORBA development sphere. Consequently, application developers have to resort to COS level solutions, such as Event/Notification Services, which are more or less message oriented rather than object oriented. In COS Event/Notification Service, the publish/subscribe is modeled as replicated client/server communications. The disadvantages of this modeling are:

- The object abstraction is at a very low level. A large semantic gap has to be filled by application developers. They have to directly manipulate low level concepts and objects of client/server communications, such as consumer proxy, supplier proxy, and so forth, including direct rearrangement of their interconnections.
- Usage of tight coupling in the object model. Although the channel connection model, message format (structured, typed, etc.) and message transfer model (push/pull) are orthogonal, they are tightly coupled. A change in one part of these components will impact other parts, especially when changing structured to typed channel or typed push to typed pull.

Note

CORBA is not the only distributed object middleware that does not provide support for non-classic communication models at the same object abstraction level. For instance, within the RMI/EJB environment, instead of extending the original Java and RMI object model, a message oriented model (namely, JMS/MDB) is used.

The Publish/Subscribe Adapter (PSA) described in this chapter addresses the problems previously mentioned. PSA is mainly a programming model and a software component working on top of OMG standardized Notification Service . Therefore, PSA can be used along with third party OMG Notification Service implementations and is also interchangeable with applications which are directly built with low-level OMG Notification Service interfaces.

One of the basic functions of the PSA is to hide the details pertaining to channel connections. Typically, when designing a CORBA publish/subscribe application, the main goal is to make the application consumer object receive events from a given channel. The channel is usually specified by its channel reference or consumer admin reference. The consumer object is usually specified by its POA and object id. By using OMG Notification Service directly, the application requires multiple steps in connecting the consumer object to the channel. However, by using PSA, the application only needs a single operation to complete this connection.

To introduce the basic concept of PSA, this example shows how a typed event consumer application is coded. Assume that the typed event is defined by the IDL interface:

```
// TMN.idl: typed event definition
#pragma prefix "examples.borland.com"
module TMN {
     interface TypedEvent {
           void attributeValueChange(...);
           ...
     };
};
```

First, in order for the typed event consumer to be able to receive events, it needs to provide a servant implementation that derives from the user defined event interface skeleton, POA_TMN::TypedEvent:

```
// 1. Implement typed servant
# include "TMNEvents_s.hh"
class TMNTypedEventImpl : public POA_TMN::TypedEvent,
                          public PortableServer::RefCountServantBase
{
   public:
     void attributeValueChange(...) { ... };
     ...
};
```

Next, activate this servant on a POA:

```
int main(argc, argv)
{
     ...
     // 2. get orb and poa environment
     CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
     CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
     PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
     // 3. allocate the typed  servant
     TMNTypedEventImpl* servant = new TMNTypedEventImpl();
     // 4. activate it on poa
     poa->activate_object(servant);
```

Up to this point, the typed event application is treated as a normal typed consumer application and nothing special has been added. If this was a normal client/server example, then, the application would create an object reference from POA and pass it to clients. In any case, this example is of a publish/subscribe consumer, therefore, the application does not need to pass its reference directly to a client, which is the event

publisher. Instead, the consumer needs to connect to a given channel or consumer admin object reference.

With PSA, instead of “connecting” the consumer to the channel, you simple “subscribe” it to the channel:

```
        // 5. Somehow, this consumer is given a channel reference
        CORBA::Object_var channel = ... ;
        // 6. Get object id of the consumer servant
        PortableServer::ObjectId_var oid = poa->servant_to_id(servant);
        // 7. Narrow the POA to PSA
        PortableServerExt::PSA_var psa = PortableServerExt::PSA::_narrow(poa);
        // 8. Subscribe to the channel
        PortableServerExt::SubjectScheme scheme = {
               PortableServerExt::CHANNEL_ADDR,
               PortableServerExt::TYPED_SUBJECT,
               (const char*)"IDL:example.borland.com/TMN/TypedEvent:1.0",
               PortableServerExt::PUSH_EVENT };
        psa->subscribe(scheme, channel, oid, CORBA::NameValuePairSeq());
        // 9. Consumer working loop
        poa->the_POAManager()->activate();
        orb->run();
    }
```

As shown in the code, the application only needs to create the typed servant implementation with PSA. The application does not need to have the CosTypedNotifyComm::TypedPushConsumer servant to support get_typed_consumer(). Also, notice that the subscribe is one step procedure instead of multiple (six steps) operations to make a “connect.”

Here is the Java code example equivalence:

```
import com.inprise.vbroker.PortableServerExt.*;
// 1. Implement typed servant
public class TMNTypedEventImpl : extend TMN.TypedEventPOA,
{
    public void attributeValueChange(...) { ... }
};
public class TypedPushConsumerImpl
{
    public static void main(String[] args)
    {
       ...
       // 2. get orb and psa environment
       org.omg.CORBA.ORB orb = ORB_init(args, null);
       org.omg.PortableServer.POA poa
         =
org.omg.PortableServer.POA.orb.resolve_initial_references("RootPOA");
       // 3. allocate the typed servant
       TMNTypedEventImpl servant = new TMNTypedEventImpl();
       // 4. activate it on root psa
       poa.activate_object(servant);
       // 5. somehow, this consumer is given a channel reference
       org.omg.CORBA.Object channel = ...;
       // 6. Get object id of the consumer servant
       org.omg.PortableServer.ObjectId oid = psa.servant_to_id(servant);

       // 7. narrow the org.omg.PortableServer.POA to com.inprise.vbroker.PSA
       PSA psa = PSA.narrow(poa);
       // 8. subscribe to the channel
       SubjectScheme scheme = new SubjectScheme(
            SubjectAddressScheme.CHANNEL_ADDR,
            SubjectInterfaceScheme.TYPED_SUBJECT,
            "IDL:example.borland.com/TMN/TypedEvent:1.0",
```

```
            SubjectDeliveryScheme.PUSH_EVENT);
        psa.subscribe(scheme, channel, oid, null);
         // 9. working loop
         poa.the_POAManager().activate();
         orb.run();
    }
}
```

This example clearly shows how the PSA works in conjunction with OMG Event/ Notification Service or Typed Event/Notification Service. More importantly, it shows how it simplifies the CORBA publish/subscribe application by shielding it from the low level notification service objects such as admins/proxies and operations.

Later in this chapter, you will see how the PSA decouples connection logic from event interface and transfer model. Connection logic, such as subscribe, in the PSA is not affected by event interface and transfer model. For instance, changing a structured consumer to a typed consumer or changing a typed consumer from push to pull, requires no change on consumer subscribe logic but only a flag change on subject scheme. These kind of changes would require major code modifications to consumer connection logic if the PSA is not used. Additionally, this chapter provides examples that covers the various application cases and show the power and usage of the PSA.

PSA reference and PSA interface IDL

PSA is an extension of POA and supports all operations defined for POA. A POA reference in VisiBroker 5.1 can be narrowed down to a PSA reference and resolve_initial_references() with “RootPOA” and “RootPSA,” which actually return the same internal reference.

This code example shows how to get root PSA.

C++

```
// getting root PSA in C++
CORBA::Object_var ref = orb->resolve_initial_references("RootPSA");
PortableServerExt::PSA_var psa = PortableServerExt::_narrow(ref);
```

Java

```
// getting root PSA in Java
// get publisher/subscriber adapter
org.omg.CORBA.Object ref = orb.resolve_initial_references("RootPOA");
PSA psa = PSAHelper.narrow(ref);
```

PSA is defined in PortableServerExt module and (in-directly) derived from PortableServer::POA.

```
module PortableServerExt {
      interface POA : PortableServer::POA {
      readonly attribute CORBA::PolicyList the_policies;
      };
      enum SubjectAddressScheme {
           SUBSCRIBE_ADMIN_ADDR,
           PUBLISH_ADMIN_ADDR,
           CHANNEL_ADDR,
           SUBJECT_ADDR
      };


      enum SubjectInterfaceScheme {
           TYPED_SUBJECT,
           UNTYPED_SUBJECT,
           STRUCTURED_SUBJECT,
           SEQUENCE_SUBJECT
     };
```

```
    enum SubjectDeliveryScheme {
         PUSH_EVENT,
         PULL_EVENT
    };
    typedef string SubjectInterfaceId;
    struct SubjectScheme {
        SubjectAddressScheme address_scheme;
        SubjectInterfaceScheme interface_scheme;
  SubjectInterfaceId  interface_id;
  SubjectDeliveryScheme delivery_scheme;
};
typedef Object     Subject;
typedef CORBA::OctetSequence PublishSubscribeDesc;
typedef PublishSubscribeDesc SubscribeDesc;
typedef PublishSubscribeDesc PublishDesc;
exception InvalidSubjectScheme { long error; };
exception InvalidSubscribeDesc { long error; };
exception InvalidPublishDesc { long error; };
exception InvalidProperties { CORBA::StringSequence names; };
exception ChannelException { string repository_id; }
// The Publisher/Subscriber Adapter
interface PSA : POA {
  // register subject observer
  SubscribeDesc subscribe(
    in SubjectScheme   the_subject_scheme,
    in Subject     the_subject,
    in PortableServer::ObjectId the_observer_id,
    in CORBA::NameValuePairSeq the_properties )
        raises( InvalidSubjectScheme,
          InvalidProperties,
          ChannelException );
  // register subject provider
  PublishDesc publish(
    in SubjectScheme  the_subject_scheme,
    in Subject   the_subject,
    in PortableServer::ObjectId the_pullable_publisher_id,
    in CORBA::NameValuePairSeq the_properties )
        raises( InvalidSubjectScheme,
          InvalidProperties,
          ChannelException );
  // unregister observer from subject
  void unsubscribe(
   in SubscribeDesc  the_subscribe_desc )
       raises( InvalidSubscribeDesc,
         ChannelException );
  // unregister (pull mode) provider
  void unpublish(
   in PublishDesc   the_publish_desc)
       raises( InvalidPublishDesc,
         ChannelException );
  // suspend subject to push into the registered
  // observer or suspend subject to pull from the
  // registered provider

  void suspend(
    in PublishSubscribeDesc  the_desc)
       raises( ChannelException );
  // resume subject to push into the registered
  // observer or resume subject to pull from the
  // registered provider.
```

```
    Void resume(
      in PublishSubscribeDesc  the_desc)
         raises( ChannelException );
    // pull (typed) event and dispatch it to a registered
    // observer.
    unsigned long pull_and_dispatch(
      in SubscribeDesc  the_subscribe_desc,
      in unsigned long  max_count,
      in boolean   block_pulling,
      in Boolean   async_dispatch)
         raises( InvalidSubscribeDesc,
           InvalidSubjectScheme,
           ChannelException );

    // pull (typed) event and accept a given visitor to
    // 'visit' the event.
    Unsigned long pull_and_visit(
      in SubscribeDesc  the_subscribe_desc,
      in unsigned long  max_count,
      in Boolean   block_pulling,
      in PortableServer::Servant the_visitor)
         raises( InvalidSubscribeDesc,
           InvalidSubjectScheme,
           ChannelException );
    Subject the_subject_addr(
      in PublishSubscribeDesc  the_desc)
         raises( InvalidSubjectScheme );
    // low level access
    Object the_proxy_addr(
      in PublishSubscribeDesc  the_desc)
         raises( InvalidSubjectScheme );
   };
   ...
  };
```

In VisiBroker (5.1 and later), all POAs are internally implemented as PSAs. Therefore, any POA reference in VisiBroker 5.1 can always be narrowed into a PSA reference.

This code example shows the narrowing POA to PSA.

C++

```
// narrowing a POA into a PSA in C++
PortableServer::POA_var poa = parent_poa->create_POA(...);
PortableServerExt::PSA_var psa = PortableServerExt::_narrow(poa);
```

Java

```
// narrowing a POA into a PSA in Java
org.omg.PortableServer.POA poa = parent_poa.create_POA(...);
com.inprise.vbroker.PortableServerExt.PSA psa
      = com.inprise.vbroker.PortableServerExt.PSAHelper.narrow(poa);
```

User examples

The following examples compare application code written with the COS notification method and PSA. The examples are:

- Structured Push Consumer
- Typed Push Consumer
- Structured Push Supplier

– Typed Push Supplier

Structured Push Consumer

The table below compares the same structured push consumer application written in notification connection method (left column) and PSA (right column). The noticeable difference are:

– PSA simplifies the connection code from three steps to one.

– Push consumers using PSA is very similar to a normal server application.

This code example shows the connection/subscribe structured consumer to a channel in **C++**:

Structured Push Consumer using Notification Service interface (basic/cpp/structPushConsumer.C

```
// implement consumer servant
class StructuredPushConsumerImpl :
 public POA_CosNotifyComm:: StructuredPushConsumer,
 Public PortableServer::RefCountServantBase
{
  public:
   void push_structured_event(...) {...}
   ...
};
using namespace CosNotifyChannelAdmin;
int main(int argc, char** argv)
{
  // get orb and poa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> resolve_initial_references( "RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  // get channel reference
  EventChannel_var channel = ...;
  // allocate the consumer servant
  StructuredPushConsumerImpl* servant = new StructuredPushConsumerImpl();
  // activate it on root poa
  poa->activate_object(servant);
  // get consumer object reference
  CORBA::Object_var obj = poa->servant_to_reference(servant);
  CosNotifyComm::StructuredPushConsumer_var
     consumer = CosNotifyComm::StructuredPushConsumer::_narrow(obj);
  // connect to channel
  // 1. get default admin
  ConsumerAdmin_var admin = channel->default_consumer_admin();
  // 2. create a proxy
  ProxyID proxy_id;
  ProxySupplier_var proxy = admin->
     obtain_notification_push_supplier( STRUCTURED_EVENT, proxy_id);
  // narrow to the stub
  StructuredProxyPushSupplier_var supplier = StructuredProxyPushSupplier::
                          _narrow(proxy);

  // 3. connect proxy supplier
  supplier-> connect_structured_push_consumer( consumer);
  // working loop
  orb->run();
}
```

Structured Push Consumer using PSA(psa/cpp/structPushConsumer.C)

```
// implement consumer servant
class StructuredPushConsumerImpl :
 public POA_CosNotifyComm::StructuredPushConsumer,
 Public PortableServer::RefCountServantBase
{
  public:void push_structured_event(...) {...}
    ...
};
// no channel type specific namespace
int main(int argc, char** argv)
{
  // get orb and psa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> Resolve_initial_references( "RootPSA");
  PortableServerExt::PSA_var psa = PortableServerExt::PSA::_narrow(obj);
  // get channel reference
  CORBA::Object_var channel = ...;
  // allocate the consumer servant
  StructuredPushConsumerImpl* servant = new StructuredPushConsumerImpl();
  // activate it on root psa
  psa->activate_object(servant);
  // get consumer object id
  PortableServer::ObjectId_var oid = poa->servant_to_id(servant);
  // subscribe to channel

  // specify the subject scheme
  PortableServerExt::SubjectScheme scheme =
  { PortableServerExt::CHANNEL_ADDR, PortableServerExt::STRUCTURED_SUBJECT,
    (const char*)"", PortableServerExt::PUSH_EVENT };
   // 1. Subscribe
   psa->subscribe(scheme, channel, oid, CORBA::NameValuePairSeq());
  // working loop
  orb->run();
}
```

This is the **Java** equivalence.

Structured push consumer using Notification service interface (basic_java/StructPushConsumerImpl.java)

```
Import org.omg.CORBA.*;
import org.omg.PortableServer.*;
Import org.omg.CosNotifyComm.*;
// implement consumer servant
public class StructuredPushConsumerImpl : extend StructuredPushConsumerPOA
{
 public void push_structured_event(...) {...}
   ...
 public static void main(String[] args){
  // get orb and poa environment
  ORB orb = ORB_init(args, null);
  Object obj = orb.resolve_initial_references( "RootPOA");
  POA poa = POA.narrow(obj);
  // get channel reference
  EventChannel channel = ...;
  // allocate the consumer servant
  StructuredPushConsumerImpl servant = New StructuredPushConsumerImpl();
  // activate it on root poa
  poa.activate_object(servant);
```

```
  // get consumer object reference
  Object ref = poa.servant_to_reference(servant);
  StructuredPushConsumer consumer = StructuredPushConsumer.narrow(ref);
  // connect to channel
  // 1. get default admin
  ConsumerAdmin admin = channel.default_consumer_admin();
  // 2. Create a proxy
  ProxyID proxy_id;
  ProxySupplier proxy
     = admin.obtain_notification_push_supplier(STRUCTURED_EVENT, proxy_id);
  // narrow to the stub
  StructuredProxyPushSupplier supplier
     = StructuredProxyPushSupplier.narrow(proxy);
  // 3. Connect proxy supplier
  supplier.Connect_structured_push_consumer(Consumer);
  // working loop
  orb.run();
}
```

Structured push consumer using PSA (psa_java/structPushConsumer. java)

```
import org.omg.CORBA.*;
Import com.inprise.vbroker.PortableServerExt.*;
// Implement consumer servant
public class StructuredPushConsumerImpl :
extend CosNotifyComm.StructuredPushConsumer
{
 public void push_structured_event(...) {...}
    ...
 public static void main(String[] args) {
  // get orb and psa environment
  ORB orb = ORB_init(args, null);
  Object obj = orb.resolve_initial_references("RootPSA");
  PSA psa = PSA.narrow(obj);
  // get channel reference
  CORBA.Object. channel = ...;
  // allocate the consumer servant
  StructuredPushConsumerImpl servant = New StructuredPushConsumerImpl();
  // activate it on root psa
  psa.activate_object(servant);
  // get consumer object id
  org.omg.PortableServer.ObjectId oid = psa.servant_to_id(servant);
  // subscribe to channel

  // specify the subject scheme
  SubjectScheme scheme = new SubjectScheme(SubjectAddressScheme.CHANNEL_ADDR,
    SubjectInterfaceScheme.STRUCTURED_SUBJECT, (const char*)"",
    SubjectDeliveryScheme.PUSH_EVENT);
  // 1. Subscribe
  psa.subscribe(scheme, channel, oid, null);
  // working loop
  orb.run();
}
```

Typed Push Consumer

The table below shows the code written with the notification connection method (left column) and PSA (right column). The noticeable difference are:

- PSA simplifies the connection code from six steps to one.
- Typed consumer application using PSA does not require the application to provide a proxy object for get_typed_consumer(). PSA transparently supplies this function.
- Typed push consumer using PSA is very similar to a normal server application.
- Typed push consumer using PSA is almost identical to the PSA structured push consumer, where the PSA shields applications from any changes of different channels.

This code example show the connection/subscribe typed consumer to a channel in **C++**:

Typed push consumer using Notification service interface (basic/cpp/typedPushConsumer.C)

```
// Implement proxy consumer servant
class TypedPushConsumerImpl :
 public POA_CosTypedNotifyComm::TypedPushConsumer,
 Public PortableServer::RefCountServantBase
{
   CORBA::Object_var _I;
  Public:TypedPushConsumerImpl(
        CORBA::Object_ptr I) : _I(
        CORBA::Object::_duplicate(I)) {}
   CORBA::Object_ptr get_typed_consumer() {
     return CORBA::Object::_duplicate(_I);
   }
   ...
};
// implement typed servant
class TMNTypedEventImpl : public POA_TMN::TypedEvent,
 public PortableServer::RefCountServantBase
{
  public:void attributeValueChange(...);
    ...
};
using namespace CosTypedNotifyChannelAdmin;
int main(int argc, char** argv)
{
  // get orb and poa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb-> resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  // get channel reference
  TypedEventChannel channel = ... ;
  // allocate the typed  servant
  TMNTypedEventImpl* typed_servant = new TMNTypedEventImpl();
  // activate it on poa
  poa->activate_object(typed_servant);
  // get its reference
  CORBA::Object_var typed_ref = poa->servant_to_reference(typed_servant);
  // connect to channel
  // 1. allocate the proxy consumer
  TypedPushConsumerImpl* servant = new TypedPushConsumerImpl(typed_ref);
  // 2. activate it on root poa
  poa->activate_object(servant);
  // 3. get consumer object reference/
  obj = poa->servant_to_reference(servant);
  CosTypedNotifyComm::TypedPushConsumer_var
     consumer = CosTypedNotifyComm::TypedPushConsumer::_narrow(obj);
```

```
  // 4. Get default admin
  TypedConsumerAdmin_var admin = channel->default_consumer_admin();
  // 5. Create a proxy
  CosNotifyChannelAdmin::ProxyID proxy_id;
  TypedProxySupplier_var proxy = admin->obtain_notification_push_supplier(
          "IDL:example.borland.com/"
          "TMN/TypedEvent:1.0", proxy_id);
  // 6. Connect proxy supplier
  proxy->connect_typed_push_consumer(consumer);
  // working loop
  orb->run();
}
```

Typed push using PSA (psa/cpp/typedPushConsumer.C)

```
// No need to implement the proxy consumer.
// PSA transparently supplies a proxy to
// support get_typed_consumer().
// implement typed servant
class TMNTypedEventImpl : public POA_TMN::TypedEvent,
 public PortableServer::RefCountServantBase
{
  public:void attributeValueChange (...) ;
    ...
};

// no channel typed specific namespace
int main(int argc, char** argv)
{
  // get orb and psa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb->Resolve_initial_references("RootPSA");
  PortableServerExt::PSA_var psa = PortableServerExt::PSA::_narrow(obj);
  // get channel reference
  CORBA::Object_var channel = ... ;
  // allocate the typed servant
  TMNTypedEventImpl* typed_servant = new TMNTypedEventImpl();
  // activate it on root psa
  psa->activate_object(typed_servant);
  // get its object id
  PortableServer::ObjectId_var oid = poa->servant_to_id(typed_servant);
  // subscribe to channel
  // specify the subject scheme
  PortableServerExt::SubjectScheme scheme = { PortableServerExt::CHANNEL_ADDR,
    PortableServerExt::TYPED_SUBJECT, (const char*)"IDL:example.borland.com"
    "TMN/TypedEvent:1.0", PortableServerExt::PUSH_EVENT };
   // 1. Subscribe
   psa->subscribe(scheme, channel, oid, CORBA::NameValuePairSeq());
  // working loop
  orb->run();
}
```

This code example show the connection/subscribe typed consumer to a channel in **Java**:

Typed push consumer using Notification service interface (basic_java/TypedPush ConsumerImpl.C)

```
import org.omg.CORBA.*;
Import org.omg.PortableServer.*;
Import org.omg.CosNotifyComm.*;
// Implement typed servant
public class TMNTypedEventImpl : extend TMN.TypedEventPOA,
{
  public void attributeValueChange(...);
    ...
}
public class TypedPushConsumerImpl : extend TypedPushConsumerPOA
{
  Object _I = null;
  public TypedPushConsumerImpl(Object I) { _I = I; }
  // Implement get_typed_consumer();
  public object get_typed_consumer({return _I; }
    ...
  public static void main(String[] args){
   // get orb and poa environment
   ORB orb = ORB_init(args, null);
   Object obj = orb.resolve_initial_references( "RootPOA");
   POA poa = POA.narrow(obj);
   // get channel reference
   TypedEventChannel channel = ... ;
   // allocate the typed  servant
   TMNTypedEventImpl typed_servant = new TMNTypedEventImpl();
   // activate it on poa
   poa.activate_object(typed_servant);

   // get its reference
   Object typed_ref = poa.servant_to_reference( typed_servant);
  // connect to channel
  // 1. Allocate the proxy consumer
  TypedPushConsumerImpl servant = New TypedPushConsumerImpl(typed_ref);
  // 2. activate it on root poa
  poa.activate_object(servant);
  // 3. get consumer object reference
  obj = poa->servant_to_reference(servant);
  TypedPushConsumer Consumer = TypedPushConsumer.narrow(obj);
  // 4. Get default admin
  TypedConsumerAdmin admin = Channel.default_consumer_admin();
  // 5. Create a proxy
  CosNotifyChannelAdmin::ProxyID proxy_id;
  TypedProxySupplier proxy = admin.Obtain_notification_push_supplier(
          "IDL:example.borland.com/"
          "TMN/TypedEvent:1.0", proxy_id);
  // 6. Connect proxy supplier
  proxy.connect_typed_push_consumer(consumer);
  // working loop
  orb.run();
}
```

Typed push consumer using PSA (psa_java/TypedPushConsumerImpl.C)

```
import org.omg.CORBA.*;
Import com.inprise.vbroker.PortableServerExt.*;
// Implement typed servant
public class TMNTypedEventImpl : extend TMN.TypedEventPOA,
{
  public void attributeValueChange(...);
    ...
}
public class TypedPushConsumerImpl
{
 // No need to implement the proxy
 // consumer. PSA transparently supplies
 // a proxy to support
 // get_typed_consumer().
 Public static void main(String args) {
  // get orb and psa environment
  ORB orb = ORB_init(args, null);
  Object obj = orb.resolve_initial_references("RootPSA");
  PSA psa = PSA.narrow(obj);
  // get channel reference
  Object channel = ... ;
  // allocate the typed servant
  TMNTypedEventImpl typed_servant = new TMNTypedEventImpl();
  // activate it on root psa
  psa.activate_object(typed_servant);
  // get its object id
  PortableServer::ObjectId oid = psa.servant_to_id(typed_servant);

   // subscribe to channel
   // specify the subject scheme
   SubjectScheme scheme = new
    (SubjectAddressScheme.CHANNEL_ADDR, SubjectInterfaceScheme.TYPED_SUBJECT,
     (const char*)"IDL:example.borland.com" "TMN/TypedEvent:1.0",
     SubjectDeliveryScheme.PUSH_EVENT);
   // 1. Subscribe
   psa.subscribe(scheme, channel, oid, null);
  // working loop
  orb.run();
}
```

These two examples clearly illustrates how PSA drastically simplifies and unifies the procedures of connecting to a notification channel for both structured and typed consumers. It also shows how PSA decouples the event format selection and connecting logic. The application code between structured and typed channel are substantially different when low level COS Notification Service is directly used. With PSA, the two examples have almost no difference on subscribing logic.

Structured and Typed Push Supplier examples

In these two examples, a typed push supplier and a structured push supplier applications are written in notification connection method (left column) and PSA (right column). The noticeable difference is that the typed push supplier using PSA is almost identical to the PSA structured push supplier. In both cases, the PSA shields the application from the different makeup of each channel.

Structured Supplier to a Channel

This code example show the connection/subscribe structured supplier to a channel in **C++**:

Structured push supplier using Notification service interface (basic_cpp/structPushSupplier.C)

```
Using namespace CosNotifyChannelAdmin;
int main(int argc, char** argv)
{
  // get orb and poa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb->Resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  // get channel reference
  EventChannel channel = ... ;
  // connect to channel
  // 1. Get default admin
  ConsumerAdmin_var admin = Channel->default_supplier_admin();
  // 2. Create a proxy
  ProxyID proxy_id;
  ProxyConsumer_var proxy = admin->
     obtain_notification_push_consumer(STRUCTURED_EVENT, proxy_id);
  // 3. Get the StructuredProxyConsumer
  StructuredProxyPushConsumer_var consumer
      = StructuredProxyPushConsumer::_narrow(proxy);
  // Push typed events interface
  for(;;) {consumer->push_structured_event(...);
     ...
  }
  ...
}
```

Structured push supplier using Notification service interface (basic_cpp/structPushSupplier.C)

```
int main(int argc, char** argv)
{
  // get orb and psa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb->Resolve_initial_references("RootPSA");
  PSA_var psa = PSA::_narrow(obj);

  // get channel reference
  CORBA::Object_var channel = ... ;
  // publish to channel
  // 1. Publish
  PortableServerExt::SubjectScheme scheme = { PortableServerExt::CHANNEL_ADDR,
     PortableServerExt::STRUCTURED_SUBJECT, (const char*)"",
     PortableServerExt::PUSH_EVENT };
   PortableServerExt::PublishDesc_var desc = psa->publish(scheme, channel,
     PortableServer::ObjectId(), CORBA::NameValuePairSeq());
  // 2. Get the StructuredProxyConsumer
  CORBA::Object_var obj = psa->the_subject_addr(desc);
  StructuredProxyPushConsumer_var consumer
     = StructuredProxyPushConsumer::_narrow(proxy);

  // Push typed events interface
  for(;;) {consumer->push_structured_event(...);
     ...
  }
  ...
}
```

Typed Supplier to a Channel

This code example show the connection/subscribe typed supplier to a channel in **C++**:

Typed push supplier using Notification service interface (basic_cpp/typedPushSupplier.C)

```
using namespace CosTypedNotifyChannelAdmin;
int main(int argc, char** argv)
{
  // get orb and poa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb->Resolve_initial_references("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
  // get channel reference
  TypedEventChannel channel = ... ;
  // connect to channel
  // 1. Get default admin
  TypedConsumerAdmin_var admin = Channel->default_supplier_admin();
  // 2. Create a proxy
  CosNotifyChannelAdmin::ProxyID proxy_id;
  TypedProxyConsumer_var proxy = admin->obtain_notification_push_consumer(
     "IDL:example.borland.com/" "TMN/TypedEvent:1.0", proxy_id);
  // 3. Get the <I> interface
  CORBA::Object_var obj = proxy->get_typed_consumer();
  TMN::TypedEvent_var consumer = TMN::TypedEvent::_narrow(obj);
  // Push typed events interface
  for(;;) {consumer->attributeValueChange(...);
     ...
  }
  ...
}
```

Typed push using PSA(psa_cpp/typedPushSupplier.C)

```
int main(int argc, char** argv)
{
  // get orb and psa environment
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  CORBA::Object_var obj = orb->Resolve_initial_references("RootPSA");
  PSA_var psa = PSA::_narrow(obj);

  // get channel reference
  CORBA::Object_var channel = ... ;
  // publish to channel
  // 1. Publish
 PortableServerExt::SubjectScheme scheme =
  { PortableServerExt::CHANNEL_ADDR, PortableServerExt::TYPED_SUBJECT,
    (const char*)"IDL:example.borland.com" "TMN/TypedEvent:1.0",
    PortableServerExt::PUSH_EVENT };
 PortableServerExt::PublishDesc_var desc = psa->publish(scheme, channel,
     PortableServer::ObjectId(), CORBA::NameValuePairSeq());
  // 2. Get the <I> interface
  CORBA::Object_var obj = psa->the_subject_addr(desc);
  TMN::TypedEvent_var consumer = TMN::TypedEvent::_narrow(obj);

  // Push typed events interface
  for(;;) {consumer->attributeValueChange(...);
     ...
  }
  ...
}
```

These examples illustrate that there is a noticeable difference between the code and procedure for making a connection for structured and typed supplier applications when using the Notification Service interface. More importantly, while using the PSA, the connection code and procedures were almost identical for both applications.

Note

All examples used in this chapter are condensed from shipped VisiNotify and PSA examples. These examples are located in the directory: examples/vbroker/notify/basic_<cpp|java> and examples/vbroker/notify/psa_<cpp|Java>.

Subscribe a subject using PSA

The subscribe operation in PSA allows a consumer object to attach to a notification/event source for receiving (either push or pull) event messages. This is a very broad concept which can covers all possible publish/subscribe scenarios such as:

- Connecting to an OMG notification/event channel.
- Joining a multicast group and establish the external key to internal consumer id mapping.
- Connecting to an non-IIOP message oriented middle.

PSA supports these scenarios under one single programming model regardless of the low level transport mechanism and type of the channel/message format. In this release, PSA only supports subscribe to OMG notification/event channel (all four channel types). PSA subscribe operation is defined as:

```
SubscribeDesc subscribe(
    in SubjectScheme   the_subject_scheme,
    in Subject     the_subject,
    in PortableServer::ObjectId the_observer_id,
    in CORBA::NameValuePairSeq the_properties )
         raises( InvalidSubjectScheme,
                  InvalidProperties,
                  ChannelException );
```

When PSA is used on top of COS Notification, this operation performs all low-level operations of getting consumer admin, obtaining proxy suppliers and making the connection. For subscribing to a typed subject, the PSA also creates and manages the handler proxy object internally to support the get_typed_consumer() operation and only require the application to supply the observer servant implementation that support the application specified typed <I> interface.

SubjectScheme

The first parameter to the subscribe() is SubjectScheme and is defined as:

```
struct SubjectScheme {
SubjectAddressScheme  address_scheme;
 SubjectInterfaceScheme interface_scheme;
 SubjectInterfaceId  interface_id;
 SubjectDeliveryScheme delivery_scheme;
};
```

The SubjectScheme specifies what is the subject reference's address scheme, interface scheme, interface repository id (for typed channel only), and delivery scheme.

The **address_scheme** field specifies the subject reference. For example, an address can be specified, which can be used directly for push event or an address to only do subscribe. Currently, there are three values on this field for subscribing; **SUBSCRIBE_ADMIN_ADDR**, **CHANNEL_ADDR**, and **SUBJECT_ADDR**, which indicates the subject reference to the subscribe() operation is a OMG Notification Consumer Admin, a OMG Notification Channel (or typed channel) or an event direct pushing address, respectively.

The three values for the address_scheme field allow the application to subscribe in the following manner:

- **SUBSCRIBE_ADMIN_ADDR**—The subject reference to subscribe() is a OMG Notification Consumer Admin reference, PSA simply calls obtain_<...>_supplier() on the admin to allocate a proxy on the admin and then calls connect_<...>_consumer() on the proxy. The consumer reference connected to the proxy is either null (for pull mode consumer) or a push consumer object reference created from this PSA with the observer_id parameter. For typed channels, the get_typed_consumer() and get_typed_supplier() are automatically handled by PSA.
- **CHANNEL_ADDR**—The subject reference to subscribe() is an OMG Notification Channel (or typed channel). PSA simply calls _get_default_consumer_admin() on the channel to get the default admin and then handles it as a connection through this consumer admin reference.
- **SUBJECT_ADDR**—The subject reference to subscribe() is a direct event pushing address. For example, it could be a multicast IOR, or a typed <I> interface. For any other channel than typed, it is a proxy push consumer. PSA calls _get_MyAdmin()/ _get_MyChannel()/_get_default_consumer_admin() and then handles it as a connection through consumer admin. For typed channel, this is already a push <I> interface. PSA looks into the reference for a consumer admin component (not currently supported) and handles as a connection through consumer admin.

Additionally, applications need to specify **SubjectInterfaceScheme** and **SubjectDeliveryScheme**.

For **SubjectInterfaceScheme** the valid values are:

- **TYPED_SUBJECT**—Subject uses either multicast or OMG Typed Notification Channel.
- **UNTYPED_SUBJECT**—Subject uses OMG Untyped Notification Channel.
- **STRUCTURED_SUBJECT**—Subject uses OMG Structured Notification Channel.
- **SEQUENCE_SUBJECT**—Subject uses OMG Sequence Notification Channel.

For **SubjectDeliveryScheme** the valid values are:

- **PUSH_EVENT**—Subject uses either multicast or OMG Push Notification mode (any of the four OMG event types).
- **PULL_EVENT**—Subject uses OMG Pull Notification mode (any of the four OMG event types).

For connecting to a typed channel, the repository id of <I> interface must also be specified. This repository is used as the implicit event filter.

Subject Reference, Observer ID, and Properties to Subscribe()

The second and third parameters to subscribe() are the reference of the subject and the object id of a passive consumer object. The subject reference's interpretation is specified by the SubjectScheme as the first parameter to subscribe() and has been described above. The passive consumer object id specifies which consumer object, a received event, can be dispatched to.

There are two kind of consumer objects; **passive** and **active**. All push consumers are passive consumers and all pull consumers, except for typed consumer using pull_and_dispatch(), are active.

Passive consumers need to be subscribed with a valid object id and the consumer servant should be activated or able to be activated (such as, by servant manager) in the subscribing PSA (i.e. POA). Active consumers, on the other hand, do not need a valid object id to subscribe(). In fact, PSA ignores the actual object id parameter when subscribe() is called to subscribe an active consumer. Also, active consumers do not need to be activated or able to be activated.

Examples of Subscribe()

Example

This example shows how to connect to an untyped service through channel reference as a push consumer.

C++

```
PortableServerExt::SubjectScheme scheme = {
  PortableServerExt::CHANNEL_ADDR,
  PortableServerExt::UNTYPED_SUBJECT,
  (const char*)"",
  PortableServerExt::PUSH_EVENT };
PortableServerExt::SubscribeDesc_var desc psa->subscribe(
  scheme, channel, observer_oid, CORBA::NameValuePairSeq());
```

Java

```
// Java code to connect to an untyped service through channel reference as a
push consumer
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,

 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, observer_oid, null);
```

Example

This example shows how to connect to an untyped service through channel reference as a pull consumer.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::UNTYPED_SUBJECT,
 (const char*)"",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
// Java code to connect to an untyped service through channel reference as a
push consumer
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
```

Example

This example shows how to connect to an structured service through channel reference as a push consumer.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
```

```
 PortableServerExt::STRUCTURED_SUBJECT,
 (const char*)"",
 PortableServerExt::PUSH_EVENT };
PortableServerExt::SubscribeDesc_var desc psa->subscribe(
scheme, channel, observer_oid, CORBA::NameValuePairSeq());
```

Java

```
// Java code to connect to a structured service through channel reference as a
push consumer
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, observer_oid, null);
```

Example

This example shows how to connect to an structured service through channel reference as a pull consumer.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::STRUCTURED_SUBJECT,
 (const char*)"",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
// Java code to connect to a structured service through channel reference as a
pull consumer
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
```

Example

This example shows how to connect to a typed service through channel reference as a push consumer.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL:example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PUSH_EVENT };
PortableServerExt::SubscribeDesc_var desc psa->subscribe(
scheme, channel, observer_oid, CORBA::NameValuePairSeq());
```

Java

```
// Java code to connect to an typed service through channel reference as a push
consumer
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
```

```
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "IDL:example.borland.com/TMN/TypedEvent:1.0",
 SubjectDeliveryScheme.PUSH_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, observer_oid, null);
```

Example

This example shows how to connect to a typed service through channel reference as a pull consumer.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
// Java code to connect to a typed service through channel reference as a pull
consumer
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "IDL:example.borland.com/TMN/TypedEvent:1.0",
 SubjectDeliveryScheme.PULL_EVENT);
SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
```

Subscribe Descriptor and the_subject_addr()

```
Object the_subject_addr(in PublishSubscribeDesc the_desc);
```

After a successful subscribe() operation, a subscribe descriptor is returned which encapsulates all information and mapping to make other operations on the subscription, such as unsubscribe(), suspend(), resume(). Also, this descriptor can be saved into a persistent repository and can be later loaded into the same consumer process session or a new restarted consumer process session. However, the format of this descriptor is internal to the given ORB which created it. Therefore, like the object key, a subscribe descriptor must be used by the same ORB that created it.

For a subscribed push consumer, channel will actively push events to the consumer servants activated with the specified observer ids. After a successful subscribe() operations, applications with untyped/structured/sequence pull consumers can get their pull addresses (proxy pull suppliers) from PSA's the_subject_addr() along with the subscribe descriptor. The subscribe descriptor was returned from the PSA subscribe() method, as a parameter.

Example

This example shows how to get a proxy untyped/structured/sequence pull supplier from subscribe descriptor:

C++

```
CORBA::Object_var proxy_pull_supplier = psa->the_subject_addr(the_desc);
```

Java

```
org.omg.CORBA.Object proxy_pull_supplier = psa.the_subject_addr(the_desc);
```

After narrowing this reference to a specified proxy, the application can pull event from the supplier with pull()/try_pull()/pull_structured_event() and try_pull_structured_event().

Typed pull consumer is discussed in “Support of Typed Pull Consumer” section.

Unsubscribe a Subject

PSA unsubscribe() disconnects the consumer from a connected channel and cleans up any local resource, if necessary (for multicast case, it remove the subject key to observer id mapping). If the consumer is connected to an untyped and typed channel, the PSA invokes disconnect_push/pull_supplier() to the proxy.

If the consumer is connected to structured or sequence channel, the PSA, respectively, invokes disconnect_structured_push/pull_supplier() or disconnect_sequence_push/pull_supplier().

This code example shows how to unsubscribe a subject:

```
void unsubscribe(in SubscribeDesc the_subscribe_desc)
```

Publish a Subject

Publish in PSA model is defined as an operation, which attaches a supplier object or source to a notification/event channel that provides (either push or pull) event messages.

This is a very broad concept that covers all possible publish/subscribe scenarios such as:

- Connections to a OMG notification/event channel.
- Attachments to a multicast channel. However, native UDP multicast, publish() does nothing but only creates and returns a wrapper publish descriptor.
- Connections to a non-IIOP message oriented middle.

PSA supports many publish/subscribe scenarios under one single programming model regardless the low level transport mechanism and the type of the channel/message format. Currently, the PSA only supports connect to OMG notification/event channel (all four channel types). The PSA operation to subscribe a consumer (such as, observer) to a given subject is:

```
PublishDesc publish(
        in SubjectScheme          the_subject_scheme,
  in Subject         the_subject,
        in PortableServer::ObjectId  the_provider_id,
        in CORBA::NameValuePairSeq  the_properties )
     raises( InvalidSubjectScheme,
       InvalidProperties,
       ChannelException );
```

When the publish operation is used on top of the COS Notification, it performs all operation of getting supplier admin, obtaining proxy consumers, and connecting to them. Additionally, when the publish operation used with a typed subject, PSA also calls `get_typed_consumer()` on the proxy consumers to get the <I> reference.

SubjectScheme

SubjectScheme is the first parameter to publish() and is defined as:

```
struct SubjectScheme {
 SubjectAddressScheme address_scheme;
 SubjectInterfaceScheme interface_scheme;
 SubjectInterfaceId     interface_id;
 SubjectDeliveryScheme delivery_scheme;
};
```

The SubjectScheme parameters specify what are the subject reference's address scheme, interface scheme, interface repository id (for typed channel only) and delivery scheme.

The `address_scheme` field specifies the subject reference, such as, whether it is an address that can directly push events or an address that can only subscribe. Currently, VisiBroker supports three valid values for this field, which indicates that the subject reference to the publish() operation is an OMG notification consumer admin, an OMG notification channel (or typed channel) or an event direct pushing address, respectively.

The supports three address schemes for subscribe are:

- **PUBLISH_ADMIN_ADDR**—The subject reference to publish() is an OMG notification supplier admin reference, PSA simply calls `obtain_<...>_consumer()` on the admin reference to allocate a proxy on the admin and then calls `connect_<...>_supplier()` on the proxy. The supplier reference connected to the proxy is either null (for push supplier) or a pull supplier reference create from this PSA with provider_id parameter. For typed channels, `get_typed_consumer()` operation and `get_typed_supplier()` implementation are automatically handled by PSA.
- **CHANNEL_ADDR**—The subject reference to publish() is an OMG notification channel (or typed channel). PSA simply calls `_get_default_supplier_admin()` on the channel to get the default supplier admin. It handles it as connect through this consumer admin reference.
- **SUBJECT_ADDR**—The subject reference to subscribe() is a direct event pushing address. For example, it could be a multicast IOR or a typed <I> interface. This is a trivial case. PSA simply wraps a publisher descriptor and returns.

Your application will also need to specify **SubjectInterfaceScheme** and **SubjectDeliveryScheme**.

The valid **SubjectInterfaceScheme** values are:

- **TYPED_SUBJECT**—Subject uses either multicast or OMG typed notification channel.
- **UNTYPED_SUBJECT**—Subject uses OMG untyped notification channel.
- **STRUCTURED_SUBJECT**—Subject uses OMG structured notification channel.
- **SEQUENCE_SUBJECT**—Subject uses OMG sequence notification channel.

The valid **SubjectDeliveryScheme** values are:

- **PUSH_EVENT**—Subject uses either multicast or OMG push notification mode (any of the four OMG event types).
- **PULL_EVENT**—Subject uses OMG pull notification mode (any of the four OMG event types).

To connect to a typed channel, you must also specify the repository id of <I> interface. This repository is actually used for narrowing a typed push reference returned from `get_typed_consumer()` into <I> stub, which is also used as a filtering key for both push and pull events.

Subject Reference, Provider ID, and Properties to Publish()

The subject reference's interpretation is specified by the SubjectScheme as the first parameter to publish() operation. The second and third parameters to publish() are the reference of the subject and the object id of a passive supplier (such as, a supplier) object. The passive supplier object id specifies which supplier object should be used by PSA to pull events for publishing.

There are two kind of supplier objects; passive and active. All push suppliers are active while all pull suppliers are passive.

Passive suppliers need to be published with a valid object id and the publish servant should be activated or able to be activated (for example, by the servant manager) in the publishing PSA (such as, POA). Active publishes, on the other hand, do not need a

valid object id to call publish(). In fact, PSA ignores the actual object id parameter when publish() is called to publish an active supplier. Also, active suppliers do not need to be activated or able to be activated. Active suppliers do not even need servant implementations.

Note

Active suppliers do not even need servant implementations.

Examples of publish()

Example

This example shows how to connect to an untyped service through channel reference as a push supplier using namespace PortableServerExt.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::UNTYPED_SUBJECT,
 PortableServerExt::(const char*)"",
 PortableServerExt::PUSH_EVENT };
PortableServerExt::PublishDesc_var desc psa->publish(
 Scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
Byte[] desc = psa.publish(scheme, channel, null, null);
```

As specified by the scheme, the given subject reference is actually a COS Notification Service channel reference. PSA internally performs the following operations:

- Gets the default supplier admin from this channel.
- Obtains an untyped proxy push consumer from the admin.
- Encapsulates the proxy push consumer's reference inside the returned subscribe descriptor.

Example

This example shows how connects to an untyped service through channel reference as a pull supplier using namespace PortableServerExt.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::UNTYPED_SUBJECT,
 (const char*)"",
 PortableServerExt::PULL_EVENT };
PortableServerExt::PublishDesc_var desc = psa->publish(
 scheme, channel, provider_id, CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.UNTYPED_SUBJECT,
 "",
```

```
 SubjectDeliveryScheme.PULL_EVENT);
PublishDesc desc = psa.publish(scheme, channel, provider_id, null);
```

As specified by the scheme, the given subject reference is actually a COS Notification Service channel reference. PSA performs following operations:

- Gets the default supplier admin from this channel.
- Obtains an untyped proxy pull consumer from the admin.
- Creates an pull supplier object reference from PSA with the provider_id parameter as object id.
- Connects to the proxy with this pull supplier reference.
- Encapsulates the proxy pull consumer's reference inside the returned subscribe descriptor.

Example

This example shows how to connect to an structured service through channel reference as a push supplier using namespace PortableServerExt.

C++

```
SubjectScheme scheme = {
 CHANNEL_ADDR, STRUCTURED_SUBJECT, (const char*)"", PUSH_EVENT };
PublishDesc_var desc = psa->publish(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
PublishDesc desc = psa.publish(scheme, channel, null, null);
```

As specified by the scheme, the given subject reference is actually a COS Notification Service channel reference. PSA performs following operations:

- Gets the default supplier admin from this channel.
- Obtains an structured proxy push consumer from the admin.
- Encapsulates the proxy push consumer's reference inside the returned subscribe descriptor.

Example

This example shows how C++ code connects to an structured service through channel reference as a push supplier using namespace PortableServerExt.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
 PortableServerExt::STRUCTURED_SUBJECT,
 (const char*)"",
 PortableServerExt::PULL_EVENT };

PortableServerExt::PublishDesc_var desc = psa->publish(
  scheme, channel, provider_id, CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
```

```
 SubjectInterfaceScheme.STRUCTURED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
Byte[] desc = psa.publish(scheme, channel, provider_id, null);
```

As specified by the scheme, the given subject reference is actually a COS Notification Service channel reference. PSA performs following operations:

- Gets the default supplier admin from this channel.
- Obtains a structured proxy pull consumer from the admin.
- Creates an pull supplier object reference from the PSA with the provider_id parameter as object id.
- Connects to the proxy with this pull supplier reference.
- Encapsulates the proxy pull consumer's reference inside the returned subscribe descriptor.

Example

This example shows how to connect to typed service through channel reference as a push supplier using namespace PortableServerExt.

C++

```
PortableServerExt::SubjectScheme scheme = {
  PortableServerExt::CHANNEL_ADDR,
  PortableServerExt::TYPED_SUBJECT,
  (const char*)"IDL:example.borland.com/TMN/TypedEvent:1.0",
  PortableServerExt::PUSH_EVENT };
PortableServerExt::PublishDesc_var desc psa->publish(
  scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PUSH_EVENT);
PublishDesc desc = psa.publish(scheme, channel, null, null);
```

As specified by the scheme, the given subject reference is actually a COS Notification Service channel reference. PSA performs following operations:

- Gets the default supplier admin from this channel.
- Obtains a typed proxy pull consumer from the admin.
- Calls `get_typed_consumer()` on the proxy reference to get the <I> interface.
- Connects to the proxy with this pull supplier reference.
- Encapsulates the proxy push consumer's reference and the <I> interface reference inside the returned subscribe descriptor.

Example

This example shows how to connect to typed service through channel reference as a pull supplier using namespace PortableServerExt.

C++

```
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
```

```
 PortableServerExt::PULL_EVENT };
PortableServerExt::PublishDesc_var desc = psa->publish(
       scheme, channel, provider_id, CORBA::NameValuePairSeq());
```

Java

```
SubjectScheme scheme = new SubjectScheme(
 SubjectAddressScheme.CHANNEL_ADDR,
 SubjectInterfaceScheme.TYPED_SUBJECT,
 "",
 SubjectDeliveryScheme.PULL_EVENT);
PublishDesc desc = psa.publish(scheme, channel, provider_id, null);
```

As specified by the scheme, the given subject reference is actually a COS Notification Service channel reference. PSA performs following operations:

- Gets the default supplier admin from this channel.
- Obtains a typed proxy pull consumer from the admin.
- Creates a typed pull supplier object reference from this PSA with the provider_id parameter as object id.
- Creates an internal proxy supplier implementation to return the reference from the `get_typed_supplier()` method.
- Creates a reference for this proxy implementation.
- Connects to the proxy consumer with this proxy implementation's reference.
- Encapsulates the proxy pull consumer's reference inside the returned subscribe descriptor.

Publish Descriptor and the_subject_addr()

```
Object the_subject_addr(in PublishSubscribeDesc the_desc);
```

After a successful publish() operation, a publish descriptor is returned. It contains information/mapping to implement other publish() operations, such as unpublish(), suspend(), and resume(). This descriptor can be saved into a persistent repository and reloaded into the same supplier process session or a restart of a new supplier session. However, the format of this descriptor is internal to the ORB that creates it. Therefore, like the object key, a subscribe descriptor should only be used by the same ORB.

For a published passive supplier(such as, pull), the channel will pull events from the supplier servant that is activated with the specified provider id.

After a successful publish() operation, applications with active (push) suppliers, can get push addresses (the proxy push consumers or for typed channel with the <I> interface references from PSA's the_subject_addr() using the publish descriptor. The publish descriptor is returned from the PSA publish() method, as a parameter.

Example

This example shows how to get proxy untyped/structured/sequence pull supplier from a subscribe descriptor:

C++

```
CORBA::Object_var proxy_pull_supplier = psa->the_subject_addr(the_desc);
```

Java

```
org.omg.CORBA.Object proxy_pull_supplier = psa.the_subject_addr(the_desc);
```

After narrowing this reference to a specified proxy or <I> interface stubs, applications can push events into the connected channels.

Note

Typed pull supplier is discussed in the “Support of Typed Pulling” section.

Unpublish a subject

```
void unpublish(in PublishDesc the_publish_desc)
        raises( InvalidPublishDesc,
                ChannelException );
```

The PSA unpublish() disconnects the supplier from a connected channel and cleans up any local resource. If the supplier is connected to an untyped and typed channel, the PSA invokes disconnect_push/pull_consumer() to the proxy. If it is connected to structured or sequence channel, the PSA respectively invokes disconnect_structured_push/pull_consumer() or disconnect_sequence_push/pull_consumer().

Support of Typed Pulling

One major problem of the Notification Service is dealing with typed pulling. The programming model defined by OMG makes it difficult to use. It requires a pull consumer to use a mangled “Pull <I>” interface and a pull supplier to implement a “Pull <I>” servant. Changing from typed push <I> consumer/supplier to typed “Pull<I>” consumer/supplier requires a substantial code change and refinement to the application designs. In addition, the “Pull<I>” interfaces are operation specific. For example, pulling a typed event from a channel, it requires the pull consumer to be selective on the operation associated with the event. This does not parallel with either typed push consumer nor structured pull consumer. In typed push consumer, the pushing consumers do not specify which operation should be associated with the next arrived events. In structured event case, a structured pull consumer is not selective on the type_name (a counterpart of operation in typed event) of next returned event.

The PSA resolves all issues (mentioned above). The PSA has the following unique advantages:

- Like the typed push consumers, the typed pull consumers implement original <I> interfaces instead of the mangled “Pull<I>” interfaces. This makes the PSA's model intuitive, easy to use and can use existing tools (such as, a normal IDL pre-compiler) to generate type safe code. Pull consumer applications developed with PSA are always easy to understand than using the “Pull<I>” interface.
- Like the typed push suppliers, the typed pull suppliers use original <I> interface stubs instead of implementing a type specific “Pull<I>” servant.
- The typed pull consumers are not selective on the operations for returned events. This is consistent with typed push consumer and structured pull consumer.
- The PSA supports active typed pull consumer and passive typed pull consumer to meet different application requirements.

The PSA supports three kinds of typed pulling implementation:

- Passive typed pull consumer
- Active typed pull consumer
- Typed pull supplier

Passive typed pull consumer

At the code level, the passive typed pull consumer is similar to a typed push consumer. Actually, changing a typed push consumer application into a passive typed pull consumer application requires nearly no code change. To create a passive typed consumer, a consumer object still needs to be available on the POA and requires it to be subscribed to the subject with associated object id. The only difference between passive typed pull consumer and typed push consumer is:

- For a normal typed push consumer: the typed events are asynchronously pushed into the consumer process from the channel, and then, dispatched to the push consumer object.
- For a passive typed pull consumer: the typed events are synchronously pulled back to the consumer server from the channel by the consumer application and dispatched to the passive typed consumer object by the PSA as if it is an active typed push.

Therefore, to subscribe a passive typed pull consumer, a valid object id is needed in PSA subscribe() operation. After the subscribe(), application uses PSA's pull_and_dispatch() method to pull typed event from channel and dispatches into the passive consumer. Passive typed pull consumer is designed for applications that want to use passive consumer along with the control of incoming events from consumer applications.

Example

This example is of the passive typed pull consumer.

C++

```
// (examples/vbroker/notify/psa_cpp/typedPullConsumer1.C)
// Implement an active visitor
# include "TMNEvents_s.hh"
class TMNTypedEventVisitor : public POA_TMN::TypedEvent
{
 ...
   public:
 void attributeValueChange(...) { ... }
 ...
 void qosAlarm(...) { ... }
};
int main(argc, argv)
{
   ...
   // subscribe to the channel as typed pull consumer
   PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PULL_EVENT };
   PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
   // create a visitor instance
   TMNTypedEventVisitor visitor;
    // pull and visit max 100 events using block mode.
    psa->pull_and_visit(desc, 100, (CORBA::Boolean)1, &visitor);
    ...
 }
```

Compare to pushed consumer application, the only difference is how it gets the typed event. Passive typed consumer require explicit pull (using pull_and_dispatch()) by the application using PSA, instead of blocking on the ORB run() and waiting for the channel to send event asynchronously.

The logic and procedure of a passive typed pull consumer can be summarized as:

- Write a consumer servant implementation derive from POA skeleton.
- Activate the servant on a POA and get its object id.
- Subscribe to the channel in with SubjectDeliveryScheme to be PULL_EVENT.
- Calls pull_and_dispatch() on the subscribe PSA with the subscribe descriptor as parameters.

– The pull backed events will be dispatched to specified consumer servant asynchronously.

Active typed pull consumer

For active typed pull consumer, consumer servant is not registered to a POA, neither a POA need to be activated. The replied typed events are directly visited (think about visitor pattern) by a visitor implement derived from POA_<I> servant skeleton. The visitor implementation is directly specified on each call of pull_and_visit() and does not need to associate with/registered on any POA. An active typed pull consumer is more like conventional typed pulling except it implements POA_<I> to backward visit the event instead of the “Pull<I>” stub.

Example

This example is of the active typed pull consumer.

C++

```
// (examples/vbroker/notify/psa_cpp/typedPullConsumer1.C)
// Implement an active visitor
# include "TMNEvents_s.hh"
class TMNTypedEventVisitor : public POA_TMN::TypedEvent
{
 ...
   public:
 void attributeValueChange(...) { ... }
 ...
 void qosAlarm(...) { ... }
};
int main(argc, argv)
{
   ...
   // subscribe to the channel as typed pull consumer
   PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
PortableServerExt::TYPED_SUBJECT,
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PULL_EVENT };
   PortableServerExt::SubscribeDesc_var desc = psa->subscribe(
scheme, channel, PortableServer::ObjectId(), CORBA::NameValuePairSeq());
   // create a visitor instance
   TMNTypedEventVisitor visitor;
    // pull and visit max 100 events using block mode.
    psa->pull_and_visit(desc, 100, (CORBA::Boolean)1, &visitor);
    ...
 }
```

Java

```
// (examples/vbroker/notify/psa_java/TypedPullConsumer1.java)
import com.inprise.vbroker.PortableServerExt.*;
// Implement an active visitor
class TMNTypedEventVisitor extends TMN.TypedEventPOA {
{
   public void attributeValueChange(...) { ... }
       ...
   public void qosAlarm(...) { ... }
};
public class TypedPullConsumer1 {
   public static void main(String[] args) {
```

```
        ...
        // subscribe to the channel as typed pull consumer
        SubjectScheme scheme = new SubjectScheme(
           SubjectAddressScheme.CHANNEL_ADDR,
           SubjectInterfaceScheme.TYPED_SUBJECT,
           "IDL::example.borland.com/TMN/TypedEvent:1.0",
           SubjectDeliveryScheme.PULL_EVENT };
        SubscribeDesc desc = psa.subscribe(scheme, channel, null, null);
        // create a visitor instance
         TMNTypedEventVisitor visitor = new TMNTypedEventVisitor();
         // pull and visit max 100 events using block mode.
         psa.pull_and_visit(desc, 100, true, visitor);
    }
        }
```

The logic and procedure of an active typed pull consumer can be summarized as:

- Write a visitor (servant) implementation derive from POA skeleton.
- Subscribe to the channel with SubjectDeliveryScheme to be PULL_EVENT.
- Calls pull_and_visit() on the subscribe PSA with the subscribe descriptor and a visitor instance as parameters.
- The pull backed events will be invoked on the specified visitor synchronously.

Typed pull supplier

The PSA supports typed pull supplier using the “piggybacked reflective callback” technology. The reflective callback allows pull supplier to be pulled and still issue events in the same <I> interface originally defined for push mode.

Simple reflective callback without piggyback works as:

- A pull supplier implements, instantiates, and activates a predefined typed unspecified callback handler, for example, the TypedCallback::PullEvent handler that only has one single operation, such as, the pull_typed_event(). The operation has an event receiver object reference as input parameter.
- When publishing a pull supplier, the reference of this callback handler is connected to channel.

Here are some characteristics when a channel pulls the supplier:

- The channel prepares an event receive object and gets its reference.
- The channel callbacks pull supplier's TypedCallback::PullEvent handler's pull_typed_event() operation with the event receiver reference.
- The pull supplier narrows the event receive reference to <I> interface. It directly invokes the operation defined on <I> interface to send specific events to typed channel.
- The channel gets the event from event receiver and supplies it to the consumers.

The advantage of simple reflective callback is that it does not require special support on pull supplier side ORB. The disadvantage is that each pull operation requires two remote round trips. The first round trip requires the callback from channel to supplier. The second round trip requires the callback from supplier to the event receiver.

The PSA and VisiNotify support and implement piggybacked reflective callback. Piggybacked reflective callback is a variation of a simple reflective callback with the following mechanism:

- The PSA creates an internal agent object (resides in a separate internal singleton POA) which supports pull_typed_event() method without input parameters.
- A pull supplier implements, instantiates, and activates the predefined application interface independent TypedCallback::PullEvent handler.

- When application publishes a pull supplier, a callback reference that points to the ORB internal agent and also encapsulates the callback handler reference is connected to channel. The application handler is not actually connected to channel.
- The channel callback to the agent's pull_typed_event() method has no input parameter.
- Agent resolves the real handler's reference from the object id and creates a local event receiver.
- Agent makes local call on handler's pull_typed_event() with this local event receiver reference as input parameter.
- An application can narrow this local event receiver reference to <I> and issues a typed event locally.
- An agent can unpacks the event from the local event receiver and sends it back as a reply from the channel's pull_typed_event() call.

The piggybacked reflective callback is transparent to applications. For example, the application code is independent from simple or piggybacked reflective callback. Piggybacked reflective callback needs only one round trip for each pull. However, piggybacked reflective callback requires pull supplier side ORB to support. VisiBroker PSA supports piggybacked reflective callback and VisiNotify only uses piggybacked reflective callback for efficient reason.

Example

This example is of the typed pull supplier.

C++

```
// (examples/vbroker/notify/psa_cpp/typedPullSupplier.C)
// implement the TypedCallback::PullEvent handler,
// with piggybacked double callback, this handler is called back by
// local PSA instead of by remote proxy pull consumer. Therefore,
// the event receiver is also a local object.
# include <TypedCallback_s.hh>
# include "TMNEvents_c.hh"
class PullEventHandlerImpl : public POA_TypedCallback::PullEvent,
                      public virtual PortableServer::RefCountServantBase
 {
     public:
  // on typed pulling
  void pull_typed_event(
CORBA::Object_ptr event_receiver,
CORBA::Boolean  block)
  {
   // narrow to typed stub
   TMN::TypedEvent_ptr stub
= TMN::TypedEvent::_narrow(event_receiver);
   // reflect the callback to issue an
// attributeValueChange event
   stub->attributeValueChange(...);
  }
 };
 ...
// create a supplier handler servant to activate it on the PSA
PullEventHandler* handler = new PullEventHandlerImpl;
psa->activate_object_with_id(handler_id, handler);
// publish to the channel as typed pull supplier with the handler_id
// but the real <I> interface repository id.
PortableServerExt::SubjectScheme scheme = {
 PortableServerExt::CHANNEL_ADDR,
PortableServerExt::TYPED_SUBJECT,
```

```
 (const char*)"IDL::example.borland.com/TMN/TypedEvent:1.0",
 PortableServerExt::PULL_EVENT };
PortableServerExt::SubscribeDesc_var desc = psa->publish(
scheme, channel, handler_id, CORBA::NameValuePairSeq());
 // activate the PSA and wait for pulling.
 psa->the_POAManager()->activate();
 orb->run();
```

Java

```
// (examples/vbroker/notify/psa_java/TypedPullSupplier.java)
 import com.inprise.vbroker.PortableServerExt.*;
// Implement the TypedCallback::PullEvent handler,
// with piggybacked double callback, this handler is called back by
// local PSA instead of by remote proxy pull consumer. Therefore,
// the event receiver is also a local object.
class PullEventHandler
         extends com.borland.vbroker.TypedCallback.PullEventPOA {
      ...
    public void pull_typed_event(
            org.omg.CORBA.Object event_receiver,
            Boolean   block) {
      // narrow to typed stub
      TMN.TypedEvent stub = TMN.TypedEventHelper.narrow(event_receiver);
      // reflect the callback to issue an attributeValueChange event
      stub.attributeValueChange(...);
    }
 }
 public class TypedPullSupplier {
    ...
    public static void main(String[] args) {
       ...
      // create a supplier handler servant to activate it on the PSA
      PullEventHandler handler = new PullEventHandler();
      psa.activate_object_with_id(handler_id, handler);
      // publish to the channel as typed pull supplier with the handler_id
      // but the real <I> interface repository id.
      SubjectScheme scheme = new SubjectScheme(
         SubjectAddressScheme.CHANNEL_ADDR,
         SubjectInterfaceScheme.TYPED_SUBJECT,
         "IDL::example.borland.com/TMN/TypedEvent:1.0",
         SubjectDeliveryScheme.PULL_EVENT);
      SubscribeDesc desc = psa.publish(scheme, channel, handler_id, null);
       // activate the PSA and wait for pulling.
      psa.the_POAManager()Activate();
       orb->run();
    }
 }
```

The logic and procedure of a typed pull supplier can be summarized as:

- Write a TypedCallback::PullEvent supplier servant implementation from POA skeleton. The pull_typed_event() operation of this servant using reflective callback to generate typed event using the original IDL interface stub.
- Activate the PullEvent servant on a POA and get its object id.
- Publish this callback to the channel in with SubjectDeliveryScheme to be PULL_EVENT and the object id as publish() parameters.
- Active the POA to handle pull requests.

Additional topics and summary

This section contains miscellaneous information pertaining to the PSA.

ChannelException

Most PSA operations, including exceptions for the the_subject_addr() and the_proxy_addr(), can raise PortableServerExt::ChannelException. This exception has a string member that is the repository of low level CORBA User exception. For example, when calling suspend() twice while using a given push consumer subscribe describer as parameter, you will get a ChannelException with its repository_id member as being “IDL:omg.org/CosNotifyChannelAdmin/ConnectionAlreadyInactive.”

The intention of not declaring a PSA operation to raise Notification Service exception is to have the PSA framework generic. Although the current implementation of the PSA working in conjunction with OMG Notification Service or Typed Notification Service, it is straightforward and extends the support to cover other publish/subscribe infrastructure such as multicast.

Setting Notification Service QoS in PSA

The first approach for setting a QoS policy on a connected proxy within a PSA application is using property parameters of subscribe() and publish(). This approach is not implemented in VisiBroker 5.1.

Another approach for setting a QoS policy is to directly get the proxy reference. After using subscribe/publish operation using the _proxy_addr() method , change the policy by using set_qos() on the proxy reference.

PSA Summary

This list summarizes PSA's concepts and features:

- The PSA presents an intuitive high level object abstraction for publish/subscribe systems and shield applications from low level objects such as connections, admins, and proxies.
- The PSA supports publish/subscribe as a first class subject and provides a high level programmatic model very similar to the POA model. Using PSA to develop CORBA publish/subscribe applications is similar to developing POA based client/server applications.
- The PSA decouples orthogonal objects thereby allowing applications to change objects or logic implementations, independently. For example, changing a typed consumer from push to pull requires no change to the message receiving code but requires only a flag change on subscribe. This kind of change would be a major engagement without PSA.
- The PSA lets applications use typed event pulling by using the very same IDL <I> interface originally defined for typed event pushing rather than a cumbersome error prone mangled “Pull<I>” interface.
- Typed event pulling model under PSA/VisiNotify is symmetric to structured pulling as well as other event pull models.
- The PSA automatically handles get_typed_consumer()/get_typed_supplier() and the <I> interface to proxy mapping. This largely simplifies application code of using typed event/notification service. Typed notification applications only need to implement and install the <I> interfaces observers.
- Although the PSA is a high level programming model, it preserves and allows applications to access low level features defined in OMG Notification Service, such as querying and modifying QoS.

- With a high level abstraction, the PSA does not assume the underneath message middleware is OMG Notification Service. The same PSA programming model can transparently support various multicast transports and non-OMG message middleware.

4

Setting the Quality of Service and Filters

This section discusses how to set up the notification channel using event types and configuring with Filters and QoS properties.

Properties of the Quality of Service (QoS)

The policies set with QoS allows the application to dynamically adjust the service parameters of the channel during runtime. VisiNotify specifies its own QoS policies as well as supports a subset of the OMG-specified QoS. The following are the QoS properties that VisiNotify supports:

Priority

Setting/getting of Priority QoS is supported as per the OMG standard. Priority is represented as a short value, where –32,767 is the lowest priority and 32,767 the highest. The default priority for all events is 0 (zero). Priority can be set at message/proxy/admin/channel levels. Note that setting this property on a per-ConsumerAdmin or per-proxy supplier basis has no meaning.

EventReliability

EventReliability QoS is supported as per the OMG standard. For performance reasons, each individual proxy supplier is not guaranteed to remember persistently what events it has sent to its associated consumer. Therefore, an event can be sent to a consumer more than once if the event channel crashes and restarts.

VBPersistentDbType

This property specifies the type of persistent storage being used for persisting events by the channel. VisiNotify stores persistent events either in memory mapped files or flat files depending on the value of this property. A value of (CORBA::Short)1 implies memory mapped persistency. A value of (CORBA::Short)2 implies flat file persistency.

The default is memory mapped persistency.

VBPersistentCommitSyncPolicy

The VBPersistentCommitSyncPolicy property specifies whether the channel should acknowledge a supplier only after successfully committing the event into persistent storage.

The constant values setting are:

- **True**—a channel acknowledges a supplier only after successfully committing a event into persistent storage.
- **False (default)**—a channel can acknowledge supplier (such as, return from a push() call) immediately before committing the event into persistent storage and perform a lazy commit, afterwards.

VBPersistentStorageOverflowBlockTimeout

There are times when a new event arrives in the channel and needs to be made persistent. However, there is a possibility that persistent storage is already full. To avoid this problem, the supplier can be blocked until space becomes available in persistent storage.

The VBPersistentStorageOverflowBlockTimeout property specifies how long the supplier should be blocked to wait for persistent storage to be freed up. Upon expiration of this time interval, the channel will attempt to downgrade one or more events to BestEffort to accommodate the new event (see VBPersistentOverflowDowngradePolicy below).

The default value of this property is 0, meaning that the channel will not block, but instead it immediately attempts to downgrade queued events according to VBPersistentOverflowDowngradePolicy.

VBPersistentOverflowDowngradePolicy

The VBPersistentOverflowDowngradePolicy property controls how the channel will downgrade existing events to make way for a new (persistent) event if no space can be found in persistent storage despite waiting for VBPersistentStorageOverflowBlockTimeout seconds. If an event is downgraded, this means that its EventReliability is automatically set to BestEffort, regardless of the message/channel settings.

The constant values setting are:

- **AnyOrder (default)**—Lifo is used.
- **FifoOrder**—Events in the queue are downgraded in ascending order of the time of receipt of the event.
- **LifoOrder**—The new event is downgraded.

Note

If an event by itself cannot fit into persistent storage, it is downgraded immediately.

ConnectionReliability

The ConnectionReliability is supported as per the OMG standard.

If ConnectionReliability is set to Persistent (at the appropriate channel/admin/proxy), VisiNotify attempts to recover the following:

1. All persistent channels with original policies, ids, IOR (of the channel) and all included events.
2. All persistent admins with original policies, ids, and IOR.
3. All persistent proxies with original policies, ids, IOR, attached suppliers/consumers.

If the ConnectionReliability of a proxy is not specified explicitly through set_qos(), then the default value is used for active proxies. For example, for proxy pull consumers and proxy push suppliers are to Persistent and that of passive proxies are described for vbroker.notify.channel.passiveProxyPersistenceMask.

MaxEventsPerConsumer

The MaxEventsPerConsumer is supported as per the OMG standard.

DiscardPolicy

To facilitate implementation, such as persistent storage management, only OMG's AnyOrder, FifoOrder and LifoOrder are supported.

OrderPolicy

This QoS property sets the policy used by a given proxy to order the events it has buffered for delivery (either to another proxy or a consumer). AnyOrder, FifoOrder(default) and PriorityOrder are supported.

Note

This property has no meaning if set on a per-message basis.

VBQueueLowWaterMark

After the number of pending events in a proxy supplier queue has breached the VBQueueHighWaterMark level, (this is when a number of pending events subsequently falls below this value), this proxy informs the channel(s) to take the necessary action of unblocking or speeding up pushing and pulling event into the channel. See section on Flow Control for more details.

The default value for VBQueueLowWaterMark is 32.

VBQueueHighWaterMark

When the number of pending events in a proxy supplier queue is higher than the set value, this proxy informs the channel accordingly so that the channel can take action if necessary to block or slow down the rate of pushing and pulling events into the channel. See section on Flow Control for more details.

The default value for VBQueueHighWaterMark is computed by VisiNotify and depends on the user defined setting of the channel queue size (the channel's admin property) and the VBQueueLowWaterMark setting.

VBProxyPushSupplierThreadModel

Each proxy push supplier needs a thread to push the events in its queue to the connected push consumer. This property specifies whether a proxy should use a dedicated thread or a thread pool to push events.

Valid values are “dedicated” or “pool” with “pool” being the default value. Any illegal value is silently ignored. For additional information, see the static property section on configuring the thread pool. Setting this property on the channel or consumer admin will make all sub objects inherit this value. Setting this property on the supplier admin or any other type of proxy has no meaning and will be silently ignored.

VBProxyPushSupplierQueuePreemptWaterMark

This property is used to fine-tune the thread pool behavior and is applicable only when the VBProxyPushSupplierThreadModel is set to “pool” and the thread pool is restricted to have a finite number of threads to serve the proxy push suppliers. A proxy push object picks up a thread from the thread pool to push events to the connected thread pool. If this proxy object has a lot of pending events, it may end up hogging the thread, leaving other proxies starved. To have a control over this situation, a watermark in the queue of each proxy object can be set, so that on hitting the watermark, the thread gets preempted to serve a different proxy push supplier object.

The default value is computed by VisiNotify and depends on the queue size.

VBReceivedEventsCount

Indicates the number of events received. Trying to set any value on this property by using the `set_qos` API resets the counter to 0. The actual value passed in will be ignored.

VBPendingEventsCount

This is a read only property and indicates the number of events pending in the queue.

VBDiscardedEventsCount

Indicates the number of events discarded due to queue overflow. Trying to set any value on this property by using the `set_qos` API resets the counter to 0. The actual value passed in will be ignored.

VBForwardedEventsCount

Indicates the total number of events forwarded downstream. Trying to set any value on this property by using the `set_qos` API resets the counter to 0. The actual value passed in will be ignored.

VBFilteredEventsCount

Indicates the total number of events discarded due to failed filter match. Trying to set any value on this property by using the `set_qos` API resets the counter to 0. The actual value passed in will be ignored.

Administration and Validation of QoS properties

The following interfaces and methods are supported for administration of QoS properties:

Interface CosNotification::QoSAdmin

This interface is supported by channels, supplier/consumer admins and proxy suppliers/consumers and allows clients of these objects to obtain and set the QoS properties.

However, there are some limitations on the level of support:

- If set_qos() is passed a VisiBroker-specific QoS, and the property value is bad, it is silently ignored and no exception is thrown. Exception is thrown for only OMG specified QoS.

- The propagation of QoS changes is dissipated down the channel/admin/proxy hierarchy only for OMG specified QoS and not VisiBroker-specific QoS.

Validating QoS in the header of structured events

This is not currently supported.

QoS negotiation

The following QoS negotiation APIs are not currently supported, including:

- CosNotification::QoSAdmin::validate_qos()
- CosNotifyChannelAdmin::ProxySupplier::validate_event_qos()
- CosNotifyChannelAdmin::ProxyConsumer::validate_event_qos()

Channel Admin Properties

The following interfaces are supported for the Channel Admin:

Interface CosNotification::AdminPropertiesAdmin

This interface is supported by the notification and typed notification event channels. It is used to retrieve and set the admin properties on the channel(s).

The following OMG defined properties are supported:

- MaxQueueLength
- MaxConsumers
  The Default value for MaxConsumer is 256. Its value can range from 0 to 65535)
- MaxSuppliers
  The default value for MaxSupplier is 256. Its value can range from 0 to 65535)
- RejectNewEvents

VBPersistentStorageSize

A persistent event (for instance, a channel that has EventReliability set to persistent) needs to be stored in persistent storage. This admin property allows restriction on the size of the storage space to prevent VisiNotify from overrunning the disk space. VisiNotify stores persistent events in files. This admin property specifies the maximum size of this file in kilobytes.

The default value for VBPersistentStorageSize is 1024. Its type is `CORBA::Ulong`.

Static Properties

Unlike the QoS properties, the Static properties can be set only at startup time of the Notification Service and not during the execution of the service. The Static properties are specified just like any VisiBroker ORB properties such as, using –D<property_name>=<property_value>.

The following properties are supported:

vbroker.notify.console = <Boolean>

This property controls the Notification Service to display the message, “Notification Service is ready” in the VisiBroker Console.

The supported values for the vbroker.notify.console property are:

- **True (default)**—prints the message.
- **False**—does not print the message.

vbroker.notify.listener.port = <ULong>

This is an alias for vbroker.se.iiop_tp.scm.iiop_tp.listener.port.

The default value of the vbroker.notify.listener.port property is 14100.

vbroker.notify.factory.name = <string>

The vbroker.notify.factory.name property specifies the default factory name, which is created by the Notification Service. The application can do a _bind() to obtain a reference to the factory instead of doing a resolve_initial_references().

The default value of this property is “VisiNotifyChannelFactory.”

vbroker.notify.channel.name = <string>

The vbroker.notify.channel.name property specifies the name of the default channel that is created by the Notification Service. The application can do a _bind() to obtain a reference to the default channel instead of having to explicitly create one.

The default value of this property is “default_channel.”

vbroker.notify.channel.threadMaxIdle = <ULong>

The vbroker.notify.channel.threadMaxIdle property specifies that if a channel/proxy push supplier has waited for threadMaxIdle seconds and no event arrives in the queue during this time, the channel will release the thread that waits for events. The channel will restart the thread when a new event arrives.

The default value of this property is three seconds.

vbroker.notify.enableEventQoS = <Boolean>

The vbroker.notify.enableEventQoS property specifies whether the channel should make use of event-level QoS to deliver an event. If set to True, the performance of the channel is significantly degraded.

The supported values are:

- **True** —the channel will make use of event-level QoS when delivering event, such as EventReliability.
- **False (default)**—the channel ignores event-level QoS when delivering an event. Instead, the QoS setting at the proxy/admin/channel is adopted.

vbroker.notify.dir = <string>

The vbroker.notify.dir specifies the file directory or database table name of the VisiNotify persistent storage root. If the ConnectionPersistence QoS is set at the appropriate levels, VisiNotify will persist the following objects in the repository (depending on EventReliability and ConnectionReliability QoS policy):

- events
- channels
- consumer and supplier admins
- proxies
- channel admin properties, QoS, filters

The default value of this property is "./visinotify.dir."

vbroker.notify.ir = <string>

The vbroker.notify.ir property specifies the IR to be used by VisiNotify. The string that is specified can either be an IOR or a URL string (for example, corbaloc).

The default value of this property is null. In this case, VisiNotify tries to bind to the IR using osagent.

vbroker.notify.channel.persistentStorageSize = <ULong>

The vbroker.notify.channel.persistentStorageSize property is similar to the VBPersistentStorageSize channel admin property, except that it is used only for the first time the channel is started. Subsequently, VisiNotify will retrieve the current setting from persistent storage.

The default value of this property is VBPersistentStorageSize.

vbroker.notify.channel.persistentCommitPolicy = <Boolean>

The vbroker.notify.channel.persistentCommitPolicy property is similar to VBPersistentCommitSyncPolicy, except that it is used only the first time the channel is started. Subsequently, VisiNotify will retrieve the current setting from persistent storage.

The default value of this property is VBPersistentCommitSyncPolicy.

vbroker.notify.channel.persistentOverflowBlockTimeout = <ULong>

The vbroker.notify.channel.persistentOverflowBlockTimeout property is similar to VBPersistentStorageOverflowBlockTimeout, with the exception that vbroker.notify.channel.persistentOverflowBlockTimeout is used only during the initial start up of the channel. Subsequently, VisiNotify will retrieve the current setting from persistent storage.

The default value of this property is VBPersistentStorageOverflowBlockTimeout.

vbroker.notify.channel.persistentDowngradePolicy = <ULong>

The vbroker.notify.channel.persistentDowngradePolicy property is similar to VBPersistentOverflowDowngradePolicy, with the exception of that vbroker.notify.channel.persistentDowngradePolicy is used only during the initial start up of the channel. Subsequently, VisiNotify will retrieve the current setting from persistent storage.

The valid values supported are:

- AnyOrder (0)
- FifoOrder (1)
- LifoOrder (4)

If the value is set to anything else, the channel silently adopts a value of 0 (AnyOrder).

vbroker.notify.channel.persistentEvent = <Boolean>

The vbroker.notify.channel.persistentEvent property is similar to EventReliability, with the exception that it is used only during the initial start up of the channel. Subsequently, VisiNotify will retrieve the current setting from persistent storage.

A value of True makes the channel's EventReliability to be Persistent, otherwise, it is set to BestEffort.

vbroker.notify.channel.iorFile = <string>

The vbroker.notify.channel.iorFile property specifies the filename where VisiNotify can dump the IOR of the default channel. It uses the same syntax as in 3.x version, –ior <filename> option.

The default value of this property is null.

vbroker.notify.channel.passiveProxyPersistenceMask = <Boolean>

In general, it may not be necessary to persist a passive proxy (proxy push consumer or proxy pull supplier) because after a system crashes and restart, the user of such a proxy may no longer exist.

This property is used to derive the default ConnectionReliability setting of a passive proxy, using the following setting:

- let admin's persistence setting = 1 if admin's ConnectionReliability = Persistent, else let it be 0.
- default persistence of proxy = (this property setting) && (its admin's persistence setting)

If this default persistence has value of True, the default ConnectionReliability setting of a passive proxy is set to Persistent, otherwise, it is set to BestEffort.

The default value of this property is False.

vbroker.notify.channel.maxDelay = <ULong>

The vbroker.notify.channel.maxDelay property is a setting that controls the delay, (in milliseconds) that a proxy push supplier conditionally applies when delivering an event to its consumer. It also can be used to tune the performance of the channel.

The default value of this property is 2000 milliseconds. The minimum and maximum values are 20 and 2000, respectively.

vbroker.notify.threadPool.threadMax = <ULong>

This property specifies the maximum number of threads that can be present at any one time in the thread pool.

The default value of this property is 0, indicating unlimited concurrency.

vbroker.notify.threadPool.threadMin = <ULong>

This property specifies the minimum number of threads that can be present at any time in the thread pool.

The default value of this property is 0.

vbroker.notify.threadPool.threadMaxIdle = <ULong>

This property specifies the time, in seconds, that a thread in the thread pool can idle. After the idle time, the thread is garbage collected.

The default value of this property is 300 seconds.

vbroker.log.enable = <Boolean>

To see the debug log statements from this service, set this property to true. For the various source names options for debug log filtering, see the Debug Logging properties section of the *VisiBroker for C++ Developer's Guide*.

Levels of Support

The following table shows the level of support for each QoS property:

Property	Supported Values	Per Message	Per Proxy	Per Admin	Per Channel	Comments
1. General						
Priority	A value of Short	Yes	Yes	Yes	Yes	The values range from –32767 to +32767.
2. Event Persistence						
EventReliability	■ BestEffort ■ Persistent	Yes	No	No	Yes	For Persistent events, it is possible that the same event can be delivered to a consumer more than once.
VBPersistentDbType	Value of Short	No	No	No	Yes	Value of 1 (default) implies Memory mapped persistency. Value of 2 implies flat file persistency.
VBPersistentCommitSyncPolicy (extension)	■ False (default) ■ True	No	No	No	Yes	
VBPersistentStorageOverflowBlockTimeout (extension)	Any value of ULong	No	No	No	Yes	Unit:seconds
VBPersistentStorageOverflowDowngradePolicy (extension)	■ AnyOrder (0) ■ FifoOrder (1) ■ LifoOrder (4)	No	No	No	Yes	
3. Connection Persistent						

Property	Supported Values	Per Message	Per Proxy	Per Admin	Per Channel	Comments
ConnectionReliability	■ BestEffort ■ Persistent	No	Yes	Yes	Yes	
4. Queue Overflow Handling						
MaxEventsPerConsumer	Per OMG specification.	No	Yes (proxy supplier only)	Yes (consumer admin only)	Yes	
DiscardPolicy	■ AnyOrder (default) ■ FifoOrder ■ LifoOrder	No	Yes (proxy supplier only)	Yes (consumer admin only)	Yes	
5. Event Expiry						
StopTime	Not supported	Yes	No	No	No	
Timeout	Not supported	Yes	Yes	Yes	Yes	
StopTimeSupported	Not supported	No	Yes	Yes	Yes	
6. Event Delivery						
StartTime	Not supported	Yes				
StartTimeSupported	Not supported	No	Yes	Yes	Yes	
OrderPolicy	■ AnyOrder ■ FifoOrder (default) ■ PriorityOrder	No	Yes	Yes	Yes	
MaximumBatchSize	Not supported	No	Yes	Yes	Yes	
PacingInterval	Not supported	No	Yes	Yes	Yes	
7. Flow Control						
VBQueueLowWaterMark (extension)	A ULong value	No	Yes (proxy supplier only)	Yes (consumer admin only)	Yes	
VBQueueHighWaterMark (extension)	A ULong value	No	Yes (proxy supplier only)	Yes (consumer admin only)	Yes	

Event Filtering using Filter Objects

The OMG Notification Service specification defines two kinds of filters.

- Forwarding Filter
- Mapping Filter

The Forwarding filter allows events to be forwarded if it satisfies a constraint set by the clients. Thus, consumers can use forwarding filters to receive only events that interest them. The forwarding filter objects implement the `CosNotifyFilter::Filter` interface.

The Mapping filter enables consumers to change the priority and lifetime properties of events which satisfies a constraint. Mapping filter objects implements the `CosNotifyFilter::MappingFilter` interface. However, VisiNotify currently does not support mapping filters.

Filtering Events

The VisiNotify event filtering is performed on structured events, typed events and sequence of events. There is no filtering support for untyped events. For a sequence of

events, VisiNotify only filters the first event in the sequence. If the first event in the sequence does not satisfy the filter then the entire sequence is discarded.

Note

Refer to the OMG Notification Service specification, Section 2, for more information on each events (structured, typed, and sequence).

Forwarding Filter Evaluation

A filter object can be attached to a target object such as consumer/supplier proxy or consumer/supplier admin objects. Any given filter object can have a set of constraints and each constraint is expressed in the Extended Trader Constraint Language (TCL). A constraint expression either evaluates to TRUE (indicating that an event satisfies the constraint) or FALSE (indicating otherwise).

As long as one of the constraints is set to TRUE, the filter object will forward the event immediately. An event is discarded if the target object has its attached filters are set to FALSE. For more information about writing constraint expressions see Writing filter constraint expressions, and for more information about Extended TCL see Extended Trader Constraint Language (Extended TCL).

When a forwarding filter object is attached to an admin object, then all proxy objects associated with the admin object applies the forwarding filter. If there are no filters applied to a proxy or admin object then all events received are forwarded to the next delivery point.

When filters are attached to an admin object along with its proxies, then event forwarding depends on whether the admin object was created with **AND** or **OR** semantics. An admin object created with AND semantics implies that an event must pass both admin and its proxy filters. An admin object created with OR semantics implies that events must pass either admin or its proxy filters.

You can create a consumer admin by invoking `new_for_consumers()` on the channel and pass the value `AND_OP` (for AND semantics) or `OR_OP` (for OR semantics) to set the inter-filter group operator semantics on the consumer admin object. Likewise, you can create a supplier admin by invoking `new_for_suppliers()`. Calling `default_consumer_admin()` or `default_supplier_admin()` on the channel will return the default consumer admin or supplier admin, respectively, with AND semantics.

Note

Refer to the OMG Notification Service specification (Section 3.4 - The CosNotifyChannelAdmin Module) for more information on the methods used with the AND or OR semantics.

Using Forwarding Filters

Use the following steps to apply a forwarding filter:

1 **Obtain a Forwarding Filter Factory**. VisiNotify provides a default filter factory. To obtain a reference to it simply invoke the method `default_filter_factory()` on the following channel:

```
CosNotifyFilter::FilterFactory_var ffact = channel->default_filter_factory();
```

2 **Create a Forwarding Filter object**. VisiNotify only supports the Extended Trader Constraint Language as specified by the OMG Notification Service. To create a filter that specifies the constraints, simply invoke the method `create_filter(EXTENDED_TCL)` on the filter factory object obtained in Step 1.

```
CosNotifyFilter::Filter_var filter = ffact->create_filter( "EXTENDED_TCL" );
```

3 **Creating constraints.** For any given filter object a set of constraints can be associated with it. The constraint expression is written in Extended TCL.

The following shows how to create a set of constraints and simple constraint expression.

```
CosNotifyFilter::ConstraintExpSeq constraints;
constraints.length(1); // contains 1 constraint
constraints[0].constraint_expr = CORBA::string_dup( "$balance == 123.45");
```

Note

To learn more about the Extended TCL see Extended Trader Constraint Language (Extended TCL) and refer to the OMG Notification Service specification (Section 2.4 - The Default Filter Constraint Language).

4 **Adding constraints to a filter object**. To add a set of constraints simply invoke the method `add_constraints` on the filter object obtained in Step 2 passing in the set of constraints created in Step 3.

```
filter->add_constraints( constraints );
```

Note

Refer to the OMG Notification Service specification (Section 3.2.1 - The Filter Interface) to learn more about other operations such as modifying or obtaining constraints from a filter object.

5 **Adding a filter to a target object.** The target object can be an admin object or a proxy object. The creation of the target object is required before the filter object is attached to it. This example shows a structured push supplier proxy:

```
// create a structured push supplier proxy
CosNotifyChannelAdmin::ProxyID proxy_id;
CosNotifyChannelAdmin::ProxySupplier_var proxy
= admin->obtain_notification_push_supplier
CosNotifyChannelAdmin::STRUCTURED_EVENT, proxy_id);
CosNotifyChannelAdmin::StructuredProxyPushSupplier_var supplier
= CosNotifyChannelAdmin::StructuredProxyPushSupplier::_narrow(proxy);
```

To attach the filter object to a target object simply invoke `add_filter` on the target object. The `add_filter` operation accepts a filter object and returns a filter id unique to the particular target object. This example shows `add_filter` being invoked on a structured push supplier proxy and is passed a filter object created in Step 2.

```
CORBA::Long filter_id;
Filter_id = supplier->add_filter( filter );
```

Note

Refer to the OMG Notification Service specification (Section 3.2.4 - The FilterAdmin Interface) to learn more about other operations such as modifying or obtaining filters from a target object.

Forwarding Filter Limitation

VisiNotify currently does not support the following filter object methods:

- attach_callback
- detach_callback
- get_callbacks

Note

Refer to the OMG Notification Service specification (Section 2.6 - Sharing Subscriptions and Section 2.6.5 - Obligations on Filter Objects) to learn more about these methods and sharing subscriptions.

Writing Filter Constraint Expressions

Overview

A constraint expression is a boolean expression (that is, it evaluates with either TRUE or FALSE). A constraint expression typically refers to event data, which also includes filterable data that the application is most likely to base filtering decisions.

Contents of a structured event

A structured event is defined in `CosNotification.idl` as follows:

```
...
typedef string Istring;
typedef Istring PropertyName;
typedef any PropertyValue;

struct Property {
  PropertyName name;
  PropertyValue value;
};
typedef sequence<Property> PropertySeq;

typedef PropertySeq OptionalHeaderFields;
typedef PropertySeq FilterableEventBody;

struct EventType {
  string domain_name;
  string type_name;
};
struct EventType {
  string domain_name;
  string type_name;
};
struct FixedEventHeader {
  EventType event_type;
  string event_name;
};

struct EventHeader {
  FixedEventHeader fixed_header;
  OptionalHeaderFields variable_header;
};

struct StructuredEvent {
  EventHeader header;
  FilterableEventBody filterable_data;
  any remainder_of_body;
};
...
```

Contents of a typed event

A typed event contains a sequence of name-value pairs in which the first item in the sequence refers to a `CosNotification::EventType` that contains `domain_name` referring to the name of the typed interface and `type_name` referring to the name of the operation in that interface. The remaining items in the sequence of name-value pairs are filterable data in which each item contains a name referring to an input parameter for the operation within the typed interface and the value refers to the parameter value for that operation.

For example, an application may use the following IDL for a typed event:

```
interface foo {
  void bar( in string first, in long second );
};
```

In this example, the typed event `foo::bar` is received and the second item in the sequence of name-value pairs will be named `first` paired with a string value and the third item in the sequence will be named `second` paired with a long value.

Note

For details on structured events and typed events please refer to the OMG Notification Service V1.0 specification, Section 2.2, “Structured Events,” and Section 2.7, “Filtering Typed Events.”

Extended Trader Constraint Language (Extended TCL)

The OMG Notification Service V1.0 specifies the Extended Constraint Language as the default filter constraint language. Extended TCL is based on the Trader Constraint Language (TCL) from the OMG Trading Service and in addition, has a few extensions and changes made.

Note

See the OMG Notification Service V1.0 specification, Section 2.4.1 for details on the changes made to TCL.

A constraint expression written in Extended TCL evaluates to either a `TRUE` or `FALSE` value. These two values are reserved words in TCL. The value of `TRUE` in Extended TCL is `1` and the value of `FALSE` is `0` (zero). Hence, we can have an expression like the following:

```
TRUE + TRUE
```

that will yield a result of `2`. Sub-expressions can be specified by surrounding the sub-expression with brackets like the following:

```
(TRUE + TRUE) == 2
```

Accessing event data

Extended TCL supports the means of referring to complex data types (that is, the IDL types of `struct`, `enum`, `union` and `any`) within an event. An event, is represented by a `$` (dollar sign) symbol and attributes within an event are referenced by using a `.` (period) symbol similar to the C++ or Java programming constructs used today.

For example, in order to refer to a structured event's fixed header `event_name` attribute, we would write:

```
$.header.fixed_header.event_name
```

In a typed event, if the application has an interface named `foo` that has an operation named `bar` that takes in its first parameter a string called `first`, we would refer to it by writing:

```
$.first
```

Note

When the event data does not exist or if the data types of both operands for an operation does not match (for example `'A String' == 3.14`) then the constraint will evaluate to `FALSE`.

Short-hand notation

It is possible to refer to specific reserved attributes in an event as well as filterable data by using runtime variables in Extended TCL. A runtime variable is represented by prefixing a `$` (dollar sign) symbol before the identifier name. For example, `$event_name` would actually be the same as writing `$.header.fixed_header.event_name`. When runtime variables are used, the identifier is matched with reserved attributes within an event. If the identifier is not a reserved attribute within an event then it is matched with the filterable data.

Note

Please refer to OMG Notification Service V1.0 specification, Section 2.4.5 for further details on Short-hand Notation for Filtering a Generic Event.

Positional notation

The current version of VisiNotify does not support positional notation

Equality, relational and logical operators

Extended TCL uses the same operators as those used in normal TCL plus additional operators added by the OMG Notification Service V1.0 specification.

Note

The operators in the following table evaluate to either TRUE or FALSE

Operator	Description	Example
==	Equality	($.one + $.two) == 3
!=	Inequality	($.one + $.two) != 4
<	Less than	($.one + $.two) < 3
<=	Less than or equals	($.one + $.two) <= 3
>	Greater than	($.one + $.two) > 1
>=	Greater than or equals	($.one + $.two) >= 2
in	Checks if the left operand is a simple primitive type and is contained in right operand which is a sequence of the same primitive type	$.one in $.list_of_nums
~	Substring operator to check if the left operand string is contained in the right operand string	'Notify' ~ 'VisiNotify'
exist	Test to see if an identifier exists	exist $.one
and	Logical AND	($.one == 1) and ($.two == 2)
or	Logical OR	($.one == 1) or ($.two == 2)
not	Logical NOT	not exist $.one
default	Applies to discriminated union data only. Checks if a discriminated union has a default member	Default $.myUnion

Arithmetic operators

The result type of the operators in the following table depends on the type of the operands. The strongly typed operand dictates the resultant data type.

Note

Character data can be used in arithmetic operations. A string with a length of one is also considered as a character.

Operator	Description	Example
+	Addition	$.one + $.two
-	Subtraction	$.one - $.two
*	Multiplication	$.one * $.two
/	Division	$.two / $.one

Subscript operator

Arrays and sequences can be accessed via the subscript operator `[n]`. For example, in order to access the second element of an array we would write:

```
$myArray[1]
```

Lookup for name-value pairs

Name-value pair sequences are often found in events, especially the filterable data that is a name-value pair sequence. As an example, we can access the filterable data by writing an expression such as the following:

```
$.filterable_data[2].name = = "balance" and $.filterable_data[2].value > 100)
```

Expressions like these can be lengthy, hence Extended TCL allows as to write in short-form like the following:

```
$.filterable_data(balance) > 100
```

Reserved implicit members

Extended TCL makes use of reserved member attributes for the event as well as complex data. Below is a table of the reserved member attribute names and their purpose.

Attribute	Description	Example
_length	Length for an array or sequence	$.mySequence._length
_d	Discriminator for a discriminated union	$.myUnion._d
_type_id	Unscoped IDL type name	$.myData._type_id
_repos_id	Repository ID	$.myData._repository_id

Index

S

T

U

V