VisiBroker: VisiNotify Module

A white paper by Dave Stringer, Product Director, Micro Focus

Messaging – a context for Notification

Messaging Systems, or Message Oriented Middleware, is a well established paradigm for addressing the integration problem for disparate, distributed systems. It pre-dates CORBA, although CORBA can point to earlier RPC (Remote Procedure Call) architectures to stretch its antecedence back further. There has long been debate over the relative merits of the two approaches. One part of the answer is that each has certain use cases for which it is a better fit. Another part of the answer is that there is considerable overlap between implementations of the two architectures.

The CORBA Notification Service was written to augment CORBA with asynchronous communication and expand on the earlier CORBA Event Service, focusing on one-to-many topologies and the application of filtering (content based routing). The initial driver came from the Telecoms industry. However, the Notification Service has been proven to work effectively for many other domains. CORBA subsequently introduced its own Messaging Service although much of the provisions of that specification do not implement Messaging in the MOM sense.

For many organizations, CORBA Notification is today applied as an agile and cost-effective alternative to proprietary messaging solutions.

VisiNotify

Origins

The OMG Notification Service specification – which VisiNotify implements – was created to provide a CORBA rendering of the previous generation OSI System Management specification for asynchronous events (ITU-T X.711 CMIP support for CMISE M-EVENT-REPORT and related standards). OSI System Management had been widely adopted within Telecom Network Management systems. Standardization has long been crucial in Telecoms where systems need to interconnect between nations as well as span vendors.

The role VisiNotify plays in such systems is to forward events from the periphery of the network towards the center. Asynchronous communication is the preferred mode, as it is undesirable to occupy valuable processing resources waiting upon replies. VisiNotify decouples event consumers from event producers by interposing the notification channel. This also provides a many-to-many topology, allowing management applications to transparently receive events from many sources.

The notification channel performs the key step of filtering events to be relayed to different destinations. The logic for which events need to be routed to different management applications is kept out of the event producers. The key concepts of fan-in, fan-out and transforming the separation of events from a physical or geographical one (represented as horizontal) into a function separation (represented as vertical) is illustrated below.

Figure 1 – Notification Channel transforming events from one dimension to another
The channel, depicted here as a cloud, can have internal structure. It may be composed of several instances of VisiNotify channels to optimize the filtering work and to address scalability issues when there are tens of thousands of event producers and it is impractical for each one to have a TCP connection to a single VisiNotify channel.

**Understanding the Notification Service Specification**

Since VisiNotify is an implementation of the OMG’s Notification Service specification, an explanation of the specification is in order. Its basic idea is simple: a supplier notifies a consumer of an event. This simple concept becomes obscured by the number of different interfaces introduced in the specification. CORBA uses interface definitions (expressed in IDL) to provide type safety between remote objects – as strong as a programming language provides for a procedure call. Strongly typed interfaces are rigid and inflexible, which may sound like a bad thing, but they are very safe in that it is hard to use them incorrectly. They are very important for building robust distributed systems.

The specification contains interfaces like StructuredProxyPushSupplier, the nuances of which won’t be immediately apparent. This is one of twenty four interfaces named to a particular convention:

\[
\text{InterfaceName} ::= \{\text{Structured|Sequence}\} \{\text{Proxy}\} \text{Push|Pull} \text{Supplier|Consumer}
\]

The first component of the name refers to the kind of event: Structured, Sequence or Untyped. The latter is implied by omitting that component of the InterfaceName. Note that Typed events are a different case from the other three and do not follow this naming convention. Proxy is an optional component indicating that the interface is provided by a Notification Channel that resides between a set of suppliers and a set of Consumers. For a point-to-point, supplier-consumer configuration the non-Proxy, i.e. InterfaceName omitting this component, would be used. Push or Pull refers to the causality: to whether the initiating object pushed the event or pulled it, in other words whether the event is a parameter on the operation request message or the reply message. Lastly, the Supplier|Consumer component indicates the object’s role in the notification.

This complexity isn’t as bad as it might at first appear, since a given application use-case will typically only require a few of these interfaces. An architecture with fewer, more generic, interfaces would need more exceptions to trap operation invocations made in an inappropriate context. The strong typing of the interfaces in the specification ensures that setting up the suppliers and consumers is done correctly at each step and that irrelevant operations aren’t in scope.

The simplest use-case is for an application to push events into a Notification Channel. That is, to use the ProxyPushConsumer interface provided by a Notification Channel. Using interface inheritance, this interface is structured as follows:

![UML Class Diagram for Notification IDL (part)](image-url)
Again, there appears to be a lot of complexity here but for the simplest use-case all the supplier application needs is the \texttt{push()} operation. So what are all the other operations for and why are they spread over so many inherited interfaces? The other operations address the following:

- **Event types, offers and subscriptions** – a supplier can declare a list of event types it will supply (with the \texttt{offer\_change} operation) and request a list of event types consumers have subscribed for (with the \texttt{obtain\_subscription\_types} operation). This is similar to topics and queues used in Message Queues. It is useful where there is a chain of notification channels as subscriptions and offers can propagate up and down the chain.

- **Quality of Service** – changes the behavior of the Notification Channel for queue ordering, resilience (i.e. whether to make events persistent) and delivery time constraints. QoS can be set for the overall channel but the interface here sets QoS for the specific proxy consumer and only applies to events pushed to this proxy consumer.

- **Filtering** – allows fine-grained filtering of events based on the content of the event.

- **Connection management** – connections between suppliers and consumers, and the interposed proxies, are above the GIOP connections. By explicitly making supplier/consumer notification connections, the channel is able to conserve its internal resources.

A simple event supplier can acquire a \texttt{ProxyPushConsumer} interface as a \texttt{CORBA::Object} and narrow it to \texttt{CoSEventService::PushConsumer}. By doing this it will hide a lot of operation it does not use. This allows it to be built without all the generated code for the full \texttt{ProxyPushConsumer} IDL. Further simplifications can be made by having a management agent do the navigation and setup from \texttt{ChannelAdmin} that creates the \texttt{ProxyPushConsumer}.

### Implementation

OMG specifications do not come with reference implementations. This is deliberate. It allows for a differentiation between, and innovation within, implementations. The specifications provide good guarantees of portability and interoperability to ensure that customers are not locked into one implementation.

The VisiNotify implementation is differentiated from other implementations by performance, scalability and flexibility as well as supporting additional features. The key aspects of the VisiNotify implementation are described next.

#### Object level versus ORB level

IDL is a platform neutral way for describing APIs. Any application interface that is expressed in IDL can be realized on any conformant ORB infrastructure. Such an application would have source code that is portable between ORB implementations. The Notification Service specification is expressed in IDL and a simple solution (easy to build) would implement it as a pure application, i.e. at the CORBA Object level. The VisiNotify channel is not intended to be portable across other ORBs, instead it leverages its tight association with the VisiBroker ORB to gain performance benefits not accessible to pure applications. In this sense, VisiNotify is an ORB level implementation.

The diagram below illustrates how events are passed directly between \texttt{ProxyConsumer} and \texttt{ProxySupplier} without appearing at the Object level.

Compared to an ORB level implementation, an Object level implementation will potentially suffer from several weaknesses:

- Poor performance and scalability
- Weak and intricate Typed Channel support
Impenetrable to ValueTypes as (un)marshalling requires user supplied code

J2EE incompatibility as RMI/IIOP relies heavily on ValueTypes

Performance

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 of advanced techniques are used to adjust payload alignment without de-marshalling and re-marshalling the events.

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%.

Queuing

Internally, VisiNotify maintains one queue for all events accepted by the channel. Each proxy (e.g. ProxyPushSupplier) on the downstream side maintains a queue of pointers into this main queue. When filtering occurs, the proxy’s queue will be a subset of the main queue. This means that events are not duplicated in store just because they need to be delivered to multiple consumers. This negates the need for some of the properties specified on proxies intended to govern queue length and resources (number of events and memory occupied by events).

Threads

With distributed systems it is important to get the threading model correct as there is more opportunity for threads to hang while waiting for remote responses. A threading model makes a compromise between separation of function and performance. Excessive context switching between threads and scheduling large numbers of threads can degrade performance. VisiNotify offers control over the threading model inside the notification channel. ProxyPushSuppliers may share a thread pool or they may each have a dedicated thread. If each consumer gets a constant stream of events, then dedicated threads will perform better. If the event load on consumers is light or intermittent then a thread pool will perform better.

Batching

By leveraging VisiBroker’s event buffering/batch technology, the throughput displayed by VisiNotify is substantially higher in magnitude than any Object level notification service product on the market. Event buffering/batch optimizes VisiNotify throughput. The event buffering/batch technology is different from Object level batch technology, (such as sequence events) and is fully transparent to user applications with 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 an Object level implementation, event buffering/batch is not transparent to an application. Also, only a restricted event type, namely structured event, can be sent in batches. Compared to VisiBroker’s event buffering/batch technology, event batch using Sequence Channel has no advantages. VisiNotify only provides limited support for sequence channel.

Typed Notifications

The Typed Event Channel, as defined in the OMG Typed Event/Notification Service, does not predefine event interfaces. Typed Event interfaces are defined by an application using standard OMG IDL. Invoking an operation on an application-defined Typed Event interface (denoted in the specification as the <1:1> interface) sends a Typed Event to the channel. Consumer applications also receive the Typed Event by implementing and activating an object with the same application defined interface. A robust Typed Event Channel brings many benefits to applications and application developers. With Typed Event, events are formally defined via IDL interface operations and event packing and unpacking operations utilize type safe, static stub/skeleton code instead of dynamic or manual code.

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. As a result, VisiNotify’s typed channel is significantly faster than any typed or untyped channel implementations. Note that an untyped channel will transmit an event as a CORBA::Any datatype with the inefficiencies that implies.

Since VisiNotify does not rely on the interface repository when a filter is not used, the key parameters used in calling obtain_typed_component() are not necessary 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.

Relation to “oneway” operations

An operation from a normal application IDL interface can be used to “push” a typed event so long as it has no out parameters, no inout parameters, no return value and cannot raise an application exception. The reasoning is that a Notification Channel will not know what to do with any of these if they are returned. Notification isn’t a two-way communication path. These constraints are the same constraints that apply to an IDL oneway operation – an operation that
gives rise to a protocol request message but no response message. CORBA oneway operations offer a performance improvement. Note that the PushConsumer::push operation can raise a Disconnected exception and so the supplier must wait for a possible response. For a Typed consumer, the push-like operation won’t raise exceptions. Using an oneway operation to implement the typed push will give a significant additional performance improvement.

**Accessing EJBs**

Where an EJB remote interface includes a method without return parameters, i.e. suitable as a typed event operation, it is possible to arrange for it to be the push consumer connected to a Typed Notification channel. Using VisiBroker’s java2idl tool, a CORBA IDL interface can be reverse engineered from the Java EJB Remote interface. (This conforms to the OMG Java to IDL mapping specification, colloquially called the “Reverse Mapping” specification). This approach relies on CORBA ValueTypes. VisiNotify has been engineered to provide full support for passing ValueTypes through channels.

The steps necessary to connect an EJB to a notification channel as a consumer are shown below.

The Message Sequence Diagram shows the standard protocol for setting up a consumer for a Typed Channel. Here the consumer is an EJB – or more accurately, an object delegated by an EJB. The ten messages in the diagram are summarized as:

- Messages 1-4 are the standard pattern for navigating from Notification service interface to a TypedProxyPushSupplier (because this is a “obtain supplier” message for Typed channel conveys the type of the Event Observer)
- Messages 5-6 the EJB creates the Event Observer and updates the TypedPushConsumer with its object reference
- Message 7 the EJB connects the proxy supplier to the consumer and as a consequence (Message 8) the proxy supplier gets the reference for the event observer
- Messages 9-10 events from the channel are repeatedly pushed to the event observer by calling the attribute_value_change operation (the typed push operation in this example)

![Figure 4 – UML Sequence Diagram for EJB as Typed Event Consumer](image-url)
Comparison with JMS

The preceding discussion of accessing an EJB from a Typed Notification Channel is very similar to the way in which a JMS (Java Message Service) message is delivered to an MDB (Message Driven Bean) within an EJB container. VisiNotify takes the place of the JMS. Correspondences between the concepts Notification and those in JMS exist. The more significant ones are:

- JMS supports two messaging models: Point to Point, using queues, and Publish and Subscribe, using topics. VisiNotify provides the Channel, which in turn provides Proxy Suppliers. These are analogous to both JMS Queues and JMS Topics. Note also that CORBA can support point to point messaging directly with its oneway operations (i.e. without queues).
- JMS uses Destination objects to encapsulate proprietary addressing information. VisiNotify uses CORBA Object References.
- JMS has MessageSelectors for consumers to control what messages they see. VisiNotify has Filters. Filters can be applied to each Proxy Supplier.
- A JMS MessageProducer is analogous to a Notification PushSupplier and a JMS MessageConsumer is analogous to a Notification PullConsumer.

Managing channels

As with any system design, a system utilizing VisiNotify needs to be cognizant of the performance and throughput characteristics of its components. A VisiNotify channel, even when running on a dedicated node, will have an upper bound on its throughput. This bound will be dependent on several factors: the number of consumers events are to be forwarded to; the complexity of filtering constraints to be applied to events; the requirement to persist events for reliability and the peak rate at which events are to be submitted to the channel. The problem is compounded where VisiNotify shares the resources of a node with other processes.

Where there is a need to scale a VisiNotify channel to perform beyond the capabilities provided by the resources of a single node, one solution is to split the work between multiple channels assigned to separate nodes. For example, don’t evaluate all filters or forward to all consumers from the same channel instance. Instead, share the workload. Configuring VisiNotify channels to co-operate in this way is discussed below.

When the rate at which suppliers add events to the channel exceeds the rate at which events are forwarded to consumers, the queue of pending events within the channel will inevitably grow. VisiNotify, in compliance with the Notification Service specification, will discard events once all queuing resources are used up. Even though system design parameters may make this a very unlikely scenario it will be theoretically possible, especially if other parts of the system are failing in some way. A well designed system will expect to monitor channel behavior to determine if any events are discarded.

Should a VisiNotify channel become congested the practical problem is to find out what caused the congestion. It could either be an eager supplier flooding the channel with events or a lazy consumer that is not accepting events, for which it has subscribed, fast enough. In both cases there may be a single culprit out of many connected suppliers/consumers and so that individual needs to be identified.

VisiNotify provides solutions to all of these issues.

Configuring event networks

The simplest and usual case is for a single channel to reside in between a set of event suppliers and a set of event consumers. In systems that need to be highly scalable this might not be sufficient. To accommodate the population of suppliers and consumers and the load they generate might require more resources than a single machine provides. Or the topology of the network may make it sensible to break the processing into stages.

For example, there may be many tens of thousands of suppliers and providing each with a connection to a channel may exceed limits. Or a channel that has to perform processing intensive filtering and transformation between event modes (untyped, structured, sequence and typed) may benefit from a staged design to groom the events as they pass downstream.

Two VisiNotify channels can be connected together but neither has the self awareness to initiate the connection. A third element has to perform the setup to create a chain of VisiNotify channels.

Detecting channel overflows

A computer system and the processes supported by the operating system have finite resources. They may be vast but they are still finite. For an application that deals with large volumes of data, such as some uses of VisiNotify, it will always be conceivable that available resources will become exhausted. In this situation VisiNotify will discard events that have not yet been delivered to all consumers. This is channel overflow.

Event discards will be recorded in the log file, when available, but this is after the fact. A more pro-active solution is to have a management plane to a system that monitors the state of a Notification Channel. VisiNotify uses the get_qos and set_qos operations on the channel and on individual proxies to make data on the channel’s performance available.

A number of counts are available:

- VBReceivedEventsCount indicates the number of events received since start up or since last reset.
- VBPendingEventsCount 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.
VBForwardedEventsCount indicates the total number of events forwarded downstream.

VBFilteredEventsCount indicates the total number of events discarded due to failure to match against a filter.

Counts can be reset by using the set_qos operation. The value passed in is ignored. Counts are always reset to zero.

Diagnosing congestion problems

When a channel experiences an overflow it will be for one of two causes. Either an eager supplier will be flooding the channel with more events than the system was designed for or a lazy consumer may be blocking the delivery of events. In each case it may be a single supplier or consumer out of many that is causing the problem. In order to repair the faulty component (supplier or consumer) VisiNotify logs diagnostic information.

When an event is discarded because the queue is already full, details of the most recent event are logged: Channel Id, Proxy Consumer Id, Supplier Host name, Supplier port number and Timestamp.

When a consumer is too slow responding to an event push or delays too long between event pulls then its details are logged: Channel Id, Proxy Supplier Id and Timestamp.

In a well designed and sized system, operating normally, channel overflow shouldn’t happen. When it does, due to a faulty component or a change in load, providing appropriate diagnostic information is a key step to resolving the issue.

High availability

From an end-to-end perspective, VisiNotify provides asynchronous communication even though the supplier-channel and channel-consumer interactions are locally synchronous. The price of decoupling is that the suppliers don’t know if their events were actually delivered – for many situations it is good that they don’t retain any responsibility. However, this doesn’t remove the requirement to build highly available systems from the designer.

A notification channel can use a persistent store to retain events in the event of a malfunction. This goes a long way towards guaranteeing delivery but may entail a delay while the channel process is restarted. A solution that combines high availability and good performance during a malfunction is to replicate the channel.

Replicating channels

Setting up a ProxyPushConsumer interface on a notification channel involves several steps starting from the ChannelAdmin interface. Examples will often co-locate the code to do this navigation with the code that generates the events to be pushed. A short reflection should show that, although convenient in the context of an example, this isn’t always desirable. For ten or one hundred or more suppliers to a channel, there is no need for them all to perform this step. All each of them needs is a reference to the ProxyPushConsumer interface.

The same concerns arise when replicating channels. It is possible for a supplier to acquire the ChannelAdmin interface for two channel instances, to navigate from each to get two ProxyPushConsumer references and then manage, within the supplier application, the switching to the second channel when pushing to the first raises an exception. Rather than repeat this in every supplier, it is reasonable to ask the infrastructure to handle all of this.

VisiBroker uses the Object cluster feature of VisiNaming to provide a general solution to the load balancing and failover problems. To apply this to VisiNotify, a management or initialization component can handle all of the VisiNotify setup and bind two resulting ProxyPushConsumer references into an object cluster in VisiNaming. Suppliers can then resolve the given name to obtain the (clustered) ProxyPushConsumer reference. Failure of a VisiNotify channel will be transparent to a supplier application – the exception raised when pushing to a failed channel will be caught and the invocation resent to the second channel by the VisiBroker library that is linked into the application.

About Micro Focus

Micro Focus, a member of the FTSE 250, provides innovative software that allows companies to dramatically improve the business value of their enterprise applications. Micro Focus Enterprise Application Modernization and Management software enables customers’ business applications to respond rapidly to market changes and embrace modern architectures with reduced cost and risk.