INTRODUCTION TO MESSAGING ENDPOINTS

In *Introduction to Messaging Systems*, we discussed *Message Endpoint*. This is how an application connects to a messaging system so that it can send and receive messages. As an application programmer, when you program to a messaging API such as JMS or the System.Messaging namespace, you're developing endpoint code. If you are using a commercial middleware package, most of this coding is already done for you using the libraries and tools provided by the vendor.

Send and Receive Patterns

Some endpoint patterns apply to both senders and receivers. They concern how the application relates to the messaging system in general.

Encapsulate the messaging code — In general, an application should not be aware that it is using *Messaging* to integrate with other applications. Most of the application's code should be written without messaging in mind. At the points where the application integrate with others, there should be a thin layer of code that performs the application's part of the integration.

When the integration is implemented with messaging, that thin layer of code that attaches the application to the messaging system is a *Messaging Gateway*.

Data translation — It's great when the internal data representation the sender and receiver applications use are the same, and when the message format uses that same representation as well. However, this is often not the case. Either the sender and receiver disagree on data format, or the messages use a different format (usually to support other senders and receivers). In this situation, use a *Messaging Mapper* to convert data between the application's format and the message's format.

Externally-controlled transactions — Messaging systems use transactions internally; externally, by default, each `send` or `receive` method call runs in its own transaction. However, message producers and consumers have the option of using a *Transactional Client* to control these transactions externally, which is useful when you need to batch together multiple messages or to coordinate messaging with other transactional services.

Message Consumer Patterns

Other endpoint patterns only apply to message receivers. Sending messages is easy.

There are issues involved in deciding when a message should be sent, what it should contain, and how to communicate its intent to the receiver—that's why we have the Message Construction patterns (see *Introduction to Message Construction*)—but once the message is built, sending it is easy. Receiving messages, on the other hand—that's tricky. So many of these patterns are about receiving messages.

An overriding theme in message consumption is *throttling*, which means the application controlling the rate at which it consumes messages. As discussed in the *Introduction*, a potential problem any server faces is that a high volume of client requests could overload the server. With *Remote Procedure Invocation*, the server is pretty much at the mercy of the rate that clients make calls. Likewise, with *Messaging*, the server cannot control the rate at which clients send requests—but the server can control the rate at which it processes those requests. The application does not have to receive and process the messages as rapidly as they're delivered by the messaging system; it can process them at a sustainable pace and let the extras queue up to be processed in some sort of first come, first serve basis. On the other hand, if the messages are piling up too much and the server has the resources to process more messages faster, the server can increase its message consumption throughput using concurrent message consumers. So use these patterns to let your application control the rate at which it consumes messages.

Many of these patterns come in pairs that represent alternatives. You can design an endpoint one way or the other. A single application may design some endpoints one way and some endpoints the other way, but a single endpoint can only implement one alternative. Alternatives from each pair can be combined, leading to a great number of choices for how to implement a particular endpoint.

Synchronous or asynchronous consumer — One alternative is whether to use a *Polling Consumer* or an *Event-Driven Consumer*. [JMS11, pp.68-69], [Hapner, p.13], [Dickman, p.29] Polling provides the best throttling because if the server is busy, it won't run the code to receive more messages, so the messages will queue up. Consumers that are event-driven tend to process messages as fast as they arrive, which could overload the server; but each consumer can only process one message at a time, so limiting the number of consumers effectively throttles the consumption rate.

Message assignment vs. message grab — Another alternative concerns how a handful of consumers process a handful of messages. If each consumer gets a message, they can process the messages concurrently. The simplest approach is *Competing Consumers*, where one *Point-to-Point Channel* has multiple consumers. Each one could potentially grab any message; the messaging system's implementation decides which consumer gets a message.

If you want to control this message-to-consumer matching process, use a *Message Dispatcher*. This is a single consumer that receives a message but delegates it to a performer for processing. An application can throttle message load by limiting the number of consumers/performers. Also, the dispatcher in a *Message Dispatcher* can implement explicit throttling behavior.

Accept all messages or filter — By default, any message delivered on a *Message Channel* becomes available to any *Message Endpoint* listening on that channel for messages to consume. However, some consumers may not want to consume just any message on that channel, but only wish to consume messages of a certain type or description. Such a discriminating consumer can use a *Selective Consumer* to describe what sort of message it's willing to receive. Then the messaging system will only make messages matching that description available to that receiver.

Subscribe while disconnected — An issue that comes up with *Publish-Subscribe Channel*s: What if a subscriber was interested in the data being published on a particular channel and will be again, but is currently disconnected from the network or shut down for maintenance? Will a disconnected application miss messages published while it is disconnected, even though it has subscribed? By default, yes, a subscription is only valid while the subscriber is connected.

To keep the application from missing messages published in between connections, make it a *Durable Subscriber*.

Idempotency — Sometimes the same message gets delivered more than once, either because the messaging system is not certain the message has been successfully delivered yet, or because the *Message Channel*'s quality-of-service has been lowered to improve performance. Message receivers, on the other hand, tend to assume that each message will be delivered exactly once, and tend to cause problems when they repeat processing because of repeat messages. A receiver designed as an *Idempotent Receiver* handles duplicate messages and prevents them from causing problems in the receiver application.

Synchronous or asynchronous service — Another tough choice is whether an application should expose its services to be invoked synchronously (via *Remote Procedure Invocation*) or asynchronously (via *Messaging*). Different clients may prefer different approaches; different circumstances may require different approaches. Since it's often hard to choose just one approach or the other, let's have both. A *Service Activator* connects a *Message Channel* to a synchronous service in an application so that when a message is received, the service is invoked. Synchronous clients can simply invoke the service directly; asynchronous clients can invoke the service by sending a message.

Message Endpoint Themes

Another significant theme in this chapter is difficulty using *Transactional Client* with other patterns. *Event-Driven Consumer* usually cannot externally control transactions properly, *Message Dispatcher* must be carefully designed to do so, and *Competing Consumers* that externally manage transactions can run into significant problems. The safest bet for using *Transactional Client* is with a single *Polling Consumer*, but that may not be a very satisfactory solution.

Special mention should be made of JMS-style message-driven beans, one type of Enterprise JavaBeans (EJB). [EJB20, pp.311-326], [Hapner, pp.69-74] An MDB is a message consumer that is an *Event-Driven Consumer*, a *Transactional Client* that supports J2EE distributed (e.g., XAResource) transactions, and can be dynamically pooled as *Competing Consumers*, even for a *Publish-Subscribe Channel*. This is a difficult and tedious combination to implement in one's own application code, but this functionality is provided as a ready-built feature of compatible EJB containers (such as BEA's WebLogic and IBM's WebSphere). (How is the MDB framework implemented? Essentially, the container implements a *Message Dispatcher* with a dynamicly-sized pool of reusable performers, where each performer consumes the message itself using its own session and transaction.)

Finally, keep in mind that a single *Message Endpoint* may well combine several different patterns from this chapter. A group of *Competing Consumers* may be implemented as *Polling Consumer*s that are also *Selective Consumer*s and act as a *Service Activator* on a service in the application. A *Message Dispatcher* may be an *Event-Driven Consumer* and a *Durable Subscriber* that uses a *Messaging Mapper*. Whatever other patterns an endpoint implements, it should also be a *Messaging Gateway*. So don't think of what one pattern to use, think of the combinations. That's the beauty of solving the problems with patterns.

Source: http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingEndpointsIntro.html