WHY USE MESSAGING?

Now that we know what messaging is, we should ask: Why use messaging? As with any sophisticated solution, there is no one simple answer. The quick answer is that messaging is more immediate than *File Transfer,* better encapsulated than *Shared Database,* and more reliable than *Remote Procedure Invocation*. However, that's just the beginning of the advantages that can be gained using messaging.

Specific benefits of messaging include:

- *Remote Communication.* Messaging enables separate applications to communicate and transfer data. Two objects that reside in the same process can simply share the same data in memory. Sending data to another computer is a lot more complicated and requires data to be copied from one computer to another. This means that objects have to "serializable", i.e. they can be converted into a simple byte stream that can be sent across the network. If remote communication is not needed, messaging is not needed; a simpler solution such as concurrent collections or shared memory is sufficient.

- *Platform/Language Integration.* When connecting multiple computer systems via remote communication, these systems likely use different languages, technologies and platforms, perhaps because they were developed over time by independent teams. Integrating such divergent applications can require a demilitarized zone of middleware to negotiate between the applications, often using the lowest common denominator—such as flat data files with obscure formats. In these circumstances, a messaging system can be a universal translator between the applications that works with each one's language and platform on its own terms, yet allows them to all communicate through a common messaging paradigm. This universal connectivity is the heart of the *Message Bus* pattern.
- *Asynchronous Communication.* Messaging enables a *send and forget* approach to communication. The sender does not have to wait for the receiver to receive and process the message; it does not even have to wait for the messaging system to deliver the message. The sender only needs to wait for the message to be sent, e.g. for the message to successfully be stored in the channel by the messaging system. Once the message is stored, the sender is then free to perform other work while the message is transmitted in the background. The receiver may want to send an acknowledgement or result

back to the sender, which requires another message, whose delivery will need to be detected by a callback mechanism on the sender.

- *Variable Timing.* With synchronous communication, the caller must wait for the receiver to finish processing the call before the caller can receive the result and continue. In this way, the caller can only make calls as fast as the receiver can perform them. On the other hand, asynchronous communication allows the sender to batch requests to the receiver at its own pace, and for the receiver to consume the requests at its own different pace. This allows both applications to run at maximum throughput and not waste time waiting on each other (at least until the receiver runs out of messages to process).
- *Throttling.* A problem with remote procedure calls is that too many of them on a single receiver at the same time can overload the receiver. This can cause performance degradation and even cause the receiver to crash. Asynchronous communication enables the receiver to control the rate at which it consumes requests, so as not to become overloaded by too many simultaneous requests. The adverse effect on callers caused by this throttling is minimized because the communication is asynchronous, so the callers are not blocked waiting on the receiver.
- *Reliable Communication.* Messaging provides reliable delivery that a remote procedure call (RPC) cannot. The reason messaging is more reliable than

RPC is that messaging uses a *store and forward* approach to transmitting messages. The data is packaged as messages, which are atomic, independent units. When the sender sends a message, the messaging system stores the message. It then delivers the message by forwarding it to the receiver's computer, where it is stored again. Storing the message on the sender's computer and the receiver's computer is assumed to be reliable. (To make it even more reliable, the messages can be stored to disk instead of memory; see *Guaranteed Delivery*.) What is unreliable is forwarding (moving) the message from the sender's computer to the receiver's computer, because the receiver or the network may not be running properly. The messaging system overcomes this by resending the message until it succeeds. This automatic retry enables the messaging system to overcome problems with the network such that the sender and receiver don't have to worry about these details.

- *Disconnected Operation.* Some applications are specifically designed to run disconnected from the network, yet to synchronize with servers when a network connection is available. Such applications are deployed on platforms like laptop computers, PDA's, and automobile dashboards. Messaging is ideal for enabling these applications to synchronize—data to be synchronized can be queued as it is created, waiting until the application reconnects to the network.

- *Mediation.* The messaging system acts as a mediator—as in the Mediator pattern [GoF]—between all of the programs that can send and receive messages. An application can use it as a directory of other applications or services available to integrate with. If an application becomes disconnected from the others, it need only reconnect to the messaging system, not to all of the other messaging applications. The messaging system can be used to provide a high number of distributed connections to a shared resource, such as a database. The messaging system can employ redundant resources to provide high-availability, balance load, reroute around failed network connections, and tune performance and quality of service.
- *Thread Management.* Asynchronous communication means that one application does not have to block while waiting for another application to perform a task, unless it wants to. Rather than blocking to wait for a reply, the caller can use a callback that will alert the caller when the reply arrives. (See the *Request-Reply* pattern.) A large number of blocked threads, or threads blocked for a long time, can be problematic. Too many blocked threads may leave the application with too few available threads to perform real work. If an application with some dynamic number of blocked threads crashes, when the application restarts and recovers its former state, re-establishing those threads will be difficult. With callbacks, the only threads

that block are a small, known number of listeners waiting for replies. This leaves most threads available for other work and defines a known number of listener threads that can easily be re-established after a crash.

So there are a number of different reasons an application or enterprise may benefit from messaging. Some of these are technical details that application developers relate most readily to, whereas others are strategic decisions that resonate best with enterprise architects. Which of these reasons is most important depends on the current requirements of your particular applications. They're all good reasons to use messaging, so take advantage of whichever reasons provide the most benefit to you.

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