AGGREGATING STRATEGIES: MULTIPLE CHANNELS VS. SINGLE CHANNEL

When receiving loan quotes from the bank we have similar design choices. We can have all banks submit their responses to a single response channel or we can have a separate response channel for each bank. Using a single channel reduces the maintenance burden of setting up a separate channel for each participating bank, but requires each bank reply message to include a field identifying the bank who issued the quote. If we use a single response channel, the *Aggregator* may not know how many response messages to expect unless the *Recipient List* passes this information to the *Aggregator* (we call this an Initialized Aggregator). If we use an auction-style *Publish-Subscribe Channel* the number of possible responses is unknown to the loan broker so that the *Aggregator* has to employ a completeness condition that does not depend on the total number of participants. For example, the *Aggregator* could simply wait until it has a minimum of three responses. But even that would be risky if temporarily only two banks participate. In that case, the *Aggregator* could time out and report that it received an insufficient number of responses.

Managing Concurrency

A service such as a loan broker should be able to deal with multiple clients wanting to use the service concurrently. For example, if we expose the loan broker function as a Web service or connect it to a public Web site we do not really have any control over the number of clients and we may receive hundreds or thousands of concurrent requests. We can enable the loan broker to process multiple concurrent requests using two different strategies:

- Execute multiple instances.
- A single even-driven instance

The first option maintains multiple parallel instances of the loan broker component. We can either start a new instance for each incoming request or maintain a 'pool' of active loan broker processes and assign incoming requests to the next available process (using a *Message Dispatcher*). If no process is available we would queue up the requests until a process becomes available. A process pool has the advantage that we can allocate system resources in a predictable way. For example, we can decide to execute a maximum of 20 loan broker instances. In contrast, if we started a new process for each request we could quickly choke the machine if a spike of concurrent requests arrives.

Also, maintaining a pool of running processes allows us to 'reuse' an existing process for multiple requests, saving time for process instantiation and initialization.

Because much of the 'processing' required by the loan broker is to wait for replies from external parties (the credit bureau and the banks) running many parallel processes may not be a good use of system resources. Instead, we can run a single process instance that reacts to incoming message events as they arrive. Processing an individual message (e.g. a bank quote) is a relatively simple task so that a single process may be able to service many concurrent requests. This approach uses system resources more efficiently and simplifies management of the solution due to the fact that we only have to monitor a single process instance. The potential downside is the limited scalability because we are tied to one process. Many high-volume applications use a combination of the two techniques, executing multiple parallel processes each of which can handle multiple requests concurrently.

Executing multiple concurrent requests requires us to associate each message in the system to the correct process instance. For example, it may be most convenient for a bank to send all reply messages to a fixed channel. This means that the reply channel can contain messages related to different customers' concurrent quote requests.

Therefore, we need to equip each message with a *Correlation Identifier* to identify which customer request the bank is responding to.

Three Implementations

In order to implement the loan broker example, we have three main design decisions to make: we have to select a sequencing scheme for the requests , we have to select and addressing scheme for the banks and we have to define an aggregation strategy. In addition we have to select a programming language and a messaging infrastructure. In aggregate, these individual options result in a large number of potential implementation choices. We chose to implement three representative solutions to highlight the main trade-offs between the different implementation options. The following table highlights the characteristics of each solution:

Implementation	**Sequencing**	**Addressing**	**Aggregation**	**Channel Type**	**Product / Technology**
A	Synchronous	Distribution	Channel	Web Service / SOAP	Java / Apache Axis
B	Asynchronous	Distribution	Correlation ID	Message Queue	C# / Microsoft MSMQ
C	Asynchronous	Auction	Correlation ID	Pub-Sub	TIBCO ActiveEnterprise

The first implementation uses synchronous Web Services, implemented in Java and Apache Axis.

The communication with each bank occurs over a separate HTTP channel which serves both as a request and reply channel. Therefore, the aggregation strategy is based on individual reply channels and does not require correlation. The second implementation uses an asynchronous approach with message queues. We implement it using Microsoft's MSMQ, but an implementation using JMS Queues or IBM WebSphere MQ could look very similar. The last implementation uses an Auction approach and leverages TIBCO's pub-sub infrastructure and the TIB/Integration Manager *Process Manager* tool. In option B and C, all reply messages arrive on a single channel and the implementations use *Correlation Identifier*s to associate reply messages to customer loan quote inquiries.

Source:

http://www.enterpriseintegrationpatterns.com/patterns/messaging/ComposedMessagingExample.html