Asynchronous processing in WebSphere Process Server

Pamela Fong
April 29, 2009

WebSphere® Process Server offers a powerful programming model for developing asynchronous applications. In addition to the published APIs and tools for developing asynchronous systems, WebSphere Process Server also comes with a number of built-in asynchronous messaging bindings and built-in asynchronous components such as long-running processes. This article explains various asynchronous aspects in Process Server and points out best practices along the way.

Introduction

WebSphere Process Server (hereafter called Process Server) delivers a powerful programming model for developing asynchronous applications. In addition to its published APIs and tools to develop asynchronous programs using Java, Process Server also comes with a number of built-in asynchronous messaging bindings and built-in asynchronous components.

The aim of this article is to present design concepts that allow solution practitioners to design complex asynchronous solutions using this powerful platform. The first part of the article gives an overview of the Service Component Architecture (SCA) asynchronous programming model. We will be using an SCA Java™ component implementation to illustrate fundamental aspects of the asynchronous programming model using the Java programming language.

The second part of the article delves deeper into details of various asynchronous services and bindings in Process Server and how they interact with each other. Various topics specific to asynchronous processing in Process Server will be discussed:

- Transactions
- Thread considerations
- Event sequencing
- Recovery
- Clustering considerations

This article assumes that you have prior experience developing business integration modules for WebSphere Process Server.
Asynchronous programming model

WebSphere Process Server adopts Service Component Architecture (SCA) as its programming model. SCA offers seamless integration between synchronous and asynchronous interactions. This section gives a brief overview of the asynchronous capabilities offered by the SCA Java programming model.

In SCA, a service is defined by its interface. Service interfaces are always synchronous. These interfaces can be defined using WSDL or Java. In addition, the interaction can be either one-way or request-response.

Java service implementation

When a service provider implements the service's interface, it has a choice of implementing it synchronously or asynchronously. In a Java implementation, this is done by either implementing the synchronous or asynchronous interface.

Listing 1. Service impl interface

```java
public interface ServiceImplSync {
    public Object invoke(OperationType operationType, Object input) throws ServiceBusinessException;
}

public interface ServiceImplAsync {
    public void invoke(OperationType operationType, Object input, ServiceCallback callback, Ticket ticket);
}
```

You can see in Listing 1 that when a component is implemented as an asynchronous service, it is not required to return any response upon method completion. Instead, it is given a ServiceCallback proxy object that it can use to return the response at a later time.

In addition to implementing the dynamic APIs, the service provider also has a choice of implementing a service using type-safe interfaces (Listing 2).

Listing 2. Type safe service impl interface

```java
public interface StockQuote {
    public float getQuote(String symbol) throws InvalidSymbolException;
}

public interface StockQuoteAsync {
    public void getQuoteAsync(String symbol, ServiceCallback callback, Ticket ticket);
}
```

The type safe asynchronous interface is generated by SCA based on the signature of the synchronous interface during service deployment.

If a service implementation is based on the generated type-safe asynchronous interface, it is provided with a callback proxy to return a response at a later time. We categorize such a
service as asynchronous services. Therefore, a given Java component implementation can have synchronous or asynchronous behavior depending on which interface it implements. Refer to the ServiceImplSync and ServiceImplAsync documentation for more information.

Aside from the Java component type that we have examined here, Process Server offers many pre-built component types for users to build integration solutions. Often times a component type implementation can directly influence whether a business service exhibits synchronous or asynchronous behavior. For example, a human task component implementation always exhibits asynchronous behavior, regardless of the business logic implemented by the service task.

Java service invocation

When an SCA client invokes an SCA service, the invocation can be performed synchronously or asynchronously.

A synchronous call blocks the thread until the request is completed. An asynchronous invocation hands the request to an underlying messaging system so that the client can resume processing without being blocked.

There are three flavors of asynchronous invocations:

- **One-way**: Used for the fire-and-forget invocation pattern. The client calls the operation provided via the reference and control is returned immediately. No response, exception, or fault is returned. This is done via the `invokeAsync()` call.
- **Deferred response**: This is used for a request-response asynchronous invocation pattern. The client makes a request and polls for the response at a time convenient for the client. This is done via the `invokeAsync()` call.
- **Callback**: This is yet another request-response asynchronous invocation pattern. The client implements a callback interface that is invoked by the SCA runtime when the response is ready. This is done via the `invokeAsyncWithCallback()` call.

Programming details

In SCA, a service proxy is obtained through a reference lookup. Once a service is located, it can be invoked via one of the following operations from the Service interface.

The snippet in Listing 3 shows some of the operations from the Service interface.

**Listing 3. Service interface**

```java
public interface Service {
    public Object invoke(String operationName, Object input) throws ServiceBusinessException;
    public Ticket invokeAsync(String operationName, Object input);
    public Ticket invokeAsyncWithCallback(String operationName, Object input);
    public Object invokeResponse(Ticket ticket, long timeout) throws ServiceBusinessException;
}
```
One-way invocation

One-way asynchronous invocation is done via the `invokeAsync()` API.

A one-way operation is an operation that does not return any data nor throws any checked exceptions.

In SCA, there is a difference between the fire-and-forget pattern and usage of one-way operations. If a one-way operation is invoked synchronously, the caller is blocked for the duration it takes to complete the request. In addition, any unanticipated exception that occurs at the target will be thrown back to the caller. Therefore, when using the SCA Java API to invoke a target service, you must use an asynchronous invocation to attain a true fire-and-forget behavior. This is different from a business process where a one-way invocation will return the control back to the caller before the request is completed.

Callback invocation

To make a callback invocation, the caller must implement the `onInvokeResponse()` method from the `ServiceCallback` interface (Listing 4). The callback method gets invoked when the response is available.

Listing 4. Service callback interface

```java
public interface ServiceCallback {
    public void onInvokeResponse(Ticket ticket, Object output, Exception exception);
}
```

Users should be aware that when the callback method is invoked, the call always occurs in a separate thread (for example, different from the requester's thread). As a result, even though the callback invocation always takes place within the scope of the requestor's class, it is in fact a different instance of the class.

When a callback is invoked, the argument can contain a response, a business exception, or a system exception. The only type of system exception that potentially comes back via a callback is the `ServiceExpirationRuntimeException`. This can occur when an asynchronous request or response expiration qualifier is used. Other forms of system exceptions are captured by the recovery subsystem.

A sample program that you can download, `howto.asyncCallback`, illustrates how an asynchronous invocation with callback can be implemented.

Deferred response invocation

For deferred response invocations, SCA returns a correlation ticket when the request is successfully placed to a queue. A user can retrieve the response at a later time. The response can
be retrieved in the same calling thread, or a separate thread. Users should be aware that retrieving response in the same calling thread can potentially impose a compromise in system integrity. The deferred response interaction pattern is only used when atomicity of input processing is not needed, or when the system is designed to do polling using a multi-threaded approach.

**Figure 1. A polling example using deferred response**

![Diagram showing a polling example using deferred response](image)

Figure 1 shows how a deferred response is used to perform polling:

1. Thread 1 issues `invokeAsync()` for a request-response interaction. This method puts a message into a request queue.
2. A correlation ticket is returned to Thread 1.
3. Thread 1 puts the ticket into an application database.
4. Thread 0 is notified about the arrival of a new ticket.
5. Thread 0 issues `invokeResponse()` to retrieve the response from the response queue. This retrieval can be performed using a polling style by specifying a wait time to control how long the thread should be blocked waiting for the response to arrive.
6. Thread 0 updates the status in the application database once the response has been retrieved, or after it has received an expiration notice from SCA during response retrieval.

In the above example, system integrity is maintained by grouping Steps 1, 2, and 3 in the same unit of work, and Steps 4, 5, and 6 in another unit of work. This can be achieved with a proper transaction qualifier setting on the component that represents Thread 1 and Thread 0. For an in-depth discussion on transaction qualifiers, refer to the **Transactions** section in this article.

Another common usage of the deferred response asynchronous communication pattern is to facilitate asynchronous invocations from a non-SCA component. For example, a JSP can invoke an SCA service via the standalone reference. Why is such an invocation asynchronous? Asynchronous invocation is often times used in user interface development to ensure front-end responsiveness.

When a JSP needs to invoke a SCA service asynchronously, deferred response is used because a JSP cannot be invoked via the SCA callback mechanism.
Figure 2. Using deferred response pattern from non-SCA component

Figure 2 shows how deferred response is used by non-SCA component:

1. JSP page uses standalone reference to locate a SCA service. JSP issues an `invokeAsync()` to invoke the service. This call returns a ticket after the message is put to the request queue. The JSP thread is resumed to perform other functions while the request is being processed.
2. The message is delivered to the SCA service in a different thread.
3. The SCA service puts the result to a response queue.
4. JSP issues `invokeResponse()` (mostly like triggered by an end user button click) to retrieve the message from the response queue.

A sample program that you can download, `howto.deferredResponse`, illustrates how this is done.

Asynchronous expiration qualifiers

Asynchronous expiration qualifiers let users define when a request or response is considered expired or invalid.

Request expiration

An asynchronous request expiration qualifier specifies the time a request message stays valid from the time it is put onto a request queue until it is picked up.

Response expiration

An asynchronous response expiration qualifier specifies the time when a request message is sent until the time a response message is received.

These qualifiers apply to any one of the three asynchronous invocation styles discussed earlier, but particularly useful in handling deferred responses.

Consider the JSP example in that a user issues a request asynchronously. The JSP thread got terminated before it retrieves the response. When the response becomes available, it can sit in the response queue indefinitely whereas the JSP page that holds the ticket to retrieve this response is now gone. This problem can be avoided by using an asynchronous response expiration qualifier. Specifying this qualifier enables the system to clean up orphaned response messages that get accumulated in the response queue.
Mixing invocation style and implementation type

When an SCA client invokes a SCA service, the invocation can be made either synchronously or asynchronously. This is orthogonal to how a service is implemented.

If we expand the variation between the two invocation styles and two service implementation types, we have the following four combinations as shown in Figure 3.

**Figure 3. Mixing invocation and implementation styles**

<table>
<thead>
<tr>
<th>Invocation</th>
<th>Service</th>
<th>Sync</th>
<th>Async</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>Good</td>
<td>Caller will be blocked Integrity concerns</td>
<td></td>
</tr>
<tr>
<td>Async</td>
<td>Performance overhead Acknowledgement concerns</td>
<td>Good</td>
<td></td>
</tr>
</tbody>
</table>

The combination in the quadrants Sync/Sync and Async/Async are the most natural ways of interacting with a service.

**Async-over-sync**

You should judiciously invoke a synchronous service asynchronously. An asynchronous invocation has more overhead than a synchronous invocation. Each asynchronous request message has to be serialized, delivered through the underlying messaging system, and de-serialized at the receiving thread. If reliability is desirable, the message has to be written to disk during the invocation. You can avoid most of the communication overhead if the target service is capable of handling synchronous requests and the caller chooses to invoke the target synchronously.

Nevertheless, there are legitimate scenarios where the caller does not want to be blocked waiting for a request to be completed. In this case, async-over-sync invocation is an appropriate choice in spite of the communication overhead.

Before using this pattern, be aware that any system errors encountered by the target service will not be returned to the caller when the invocation is made asynchronously. This is a major distinction between synchronous and asynchronous invocation in the SCA Programming Model. Refer to the Error handling section for more information about asynchronous error handling.

**Sync-over-async**

You need to avoid invoking an asynchronous service synchronously in almost all circumstances.

Typically, a service is implemented as an asynchronous service because it takes a long time to produce the result. Invoking such a time-consuming service in a synchronous manner will tie up valuable system resources. Even when the asynchronous service has low latency, this anti-pattern can raise transactional and scalability issues. Therefore, you need to avoid it.
Asynchrony in Process Server

The basic principle of trying to align a service invocation style based on the target service's implementation is not always straightforward in practice. It requires knowledge of whether a particular service has a synchronous or asynchronous implementation. In Process Server, the following component type implementations are always asynchronous:

- POJO that implements the ServiceAsyncImpl interface
- Long running BPEL
- Human tasks
- JMS import/export
- MQ import/export

The following component type implementation can take on synchronous or asynchronous behavior based on how it is invoked:

- Interface mediation component
- Mediation Flow component
- SCA import/export

In version 6.2 and prior releases, Web services import invocation using SOAP/JMS as the underlying transport is considered a synchronous service implementation because such a service invocation always returns the response in the same thread that invoked it.

Interactions between components

To ease the service invocation style determination, a preferred interaction style attribute is available at each service interface. This attribute can take on the value of Sync, Async, Any. For the most part, this attribute defaults to a value that matches with the corresponding component type implementation. This attribute is configurable for component types that have dual behaviors, or editable for sync components to allow async-over-sync invocations.

When invoking a service via the SCA Java API, you can query the preferred interaction style of the target service before deciding which invocation style to use for the invocation (Listing 5).

Listing 5. Method to query preferred interaction style

```java
public interface Service {
    String getPreferredInteractionStyle(OperationType operationType);
}
```

In most scenarios, however, a service is not invoked from the user code via Java API, but rather via wiring a component reference to the service component. In this case, how the service gets invoked is determined by the specific implementation of the calling component.

Most of the built-in components in Process Server make use of the preferred interaction style attributes of the target component. However, the treatment is slightly different among different
component type implementations. How this attribute is interpreted by various component types is documented in WebSphere Process Server invocation styles.

Since the time the above article was published, Mediation Flow Component has introduced an additional configuration parameter to control how invocation styles can be forced to a particular style during a service callout.

**Figure 4. Invocation configuration in the mediation flow component**

In the panel shown in Figure 4, you can set "Invocation Style" to synchronous or Asynchronous to enforce a specific invocation style, irrespective of the "Preferred Invocation Style" setting of the target service.

The service invoke primitive has a similar capability. For more information, refer to the Service Invoke mediation primitive topic in the WebSphere Integration Developer Information Center.

**Asynchronous boundaries**

Why is it necessary to understand how component interactions take place under the hood? An understanding of asynchronous boundary reveals where the transaction boundaries are during an invocation flow, as a transaction context cannot propagate through an asynchronous boundary.

So how are asynchronous boundaries determined? Typically, if Component A invokes Component B asynchronously, Component B will execute in a different thread from Component A. You can then say that there is an asynchronous hop, or asynchronous boundary between the two components.

There are exceptions to the general rule.

**Asynchronous hop reduction**

If a component invokes an asynchronous import binding (such as a JMS import) asynchronously, the "hop" between the calling component and the import binding handler is eliminated (Figure 5). In this case, the import binding handler will be invoked in the same thread as the calling component. This optimization eliminates a would-be asynchronous boundary. The same is true
between an asynchronous export (such as a JMS export) and the component service behind it. This has some implications to transaction context propagation, which we will discuss later in the Transactions section.

**Figure 5. Asynchronous hop reduction**

[Diagram showing asynchronous hop reduction]

**Sync-over-Async switch**

When an asynchronous service or asynchronous binding is invoked synchronously, SCA converts the invocation into an asynchronous invocation. This conversion allows a ticket and callback proxy to be passed to the target service so that it can return the response using another thread at a later time. This switch introduces an asynchronous boundary underneath a seemingly synchronous invocation.

For the most part, this switch happens transparently.

**Figure 6. Asynchronous switching**

[Diagram showing asynchronous switching]

The switch shown in Figure 6 produces two side-effects:

- The calling thread is blocked waiting for a response, while the target service tries to compute the result in a different thread. This pattern, when coupled with a high volume processing scenario, can lead to thread pool depletion. We will discuss this scenario in more details under Thread considerations.
- Another side-effect is that a potential duplicated message results when the transaction is turned on. Refer to the Deadlock section for more details.
You should avoid invoking asynchronous services synchronously whenever possible.

**Transactions**

A transaction is unit of activity, within which multiple updates to resources can be made atomic such that all or none of the updates are made permanent. WebSphere Process Server has all the transactional support that WebSphere Application Server offers. In addition, Process Server has additional controls to simplify configurations of transactional behavior across service interactions.

**SCA transaction qualifiers**

Process Server uses qualifiers to control transactional behavior of service invocations and component services executions. There are three extension points for transaction qualifier attachment in a business module, namely: reference, interface, and implementation.

- **Asynchronous invocation**: A reference level qualifier. This qualifier controls whether the request message is sent in the component’s transaction context (Commit), or in a separate local transaction context (Call).
- **Suspend transaction**: A reference level qualifier. This qualifier is ignored during an asynchronous invocation.
- **Join transaction**: An interface level qualifier. This qualifier is ignored during an asynchronous invocation.
- **Transaction**: An implementation level qualifier. This qualifier specifies whether the component is required to run in a global transaction context, local transaction context, or any transaction context.

For a complete listing of built-in qualifiers that comes with WebSphere Integration Developer (hereafter called Integration Developer) and their semantics, refer to the Quality of service qualifier reference section in the WebSphere Integration Developer Information Center.

A correct transaction qualifier configuration is critical to the integrity and recoverability of any Process Server modules. The complexities of how to configure these qualifiers increase in an asynchronous system. This section illustrates some typical usage scenarios and its corresponding configurations.

**Typical asynchronous usage**

Let's say C1 is a service that gets invoked asynchronously. It also invokes another service asynchronously.

When a service is invoked asynchronously, SCA receives messages from the SCA module queue and dispatches to the corresponding service implementation. By default, the message will be received in a global transaction. This transaction is propagated to the service component based on the transaction qualifier settings of the service.
In the example in Figure 7, C1 can optionally have an interface qualifier JoinTransaction specified. This qualifier controls whether the service joins the transaction propagated from the caller. Since a service invoked asynchronously cannot join the transaction started by its caller, the JoinTransaction qualifier value is ignored when a service is invoked asynchronously.

The transaction started by SCA during message receipt is propagated to C1’s implementation, regardless of the interface JoinTransaction qualifier setting.

C1 can have implementation qualifier "Transaction" set to global, local, or any. The correct setting depends on the application requirement. In most cases, a setting of global is recommended, especially when C1 needs to make outbound asynchronous invocations. The act of making outbound asynchronous invocation involves putting request messages onto an outbound queue. To ensure system integrity, the put operation (to send the request message to an outbound queue) and the get operation (when SCA removes a message from the inbound queue) is executed in the same unit of work. To achieve this, the setting of C1’s Transaction qualifier should be set to global, while the setting of Asynchronous Invocation qualifier at the reference should be set to commit.

The recommended setting in this example is also the default values generated by Integration Developer in v6.1.2 and later versions when a component service is created. In earlier releases, you manually configure these values to ensure system integrity.

**Transaction boundaries**

Starting in Integration Developer version 6.1.2, you can visualize the transaction boundary within an assembly diagram as shown in Figure 8.
Asynchronous processing in WebSphere Process Server

Figure 8. Transaction highlight in WebSphere Integration Developer

In most cases, a user can change transaction boundaries through transaction qualifier configuration to match with application usage. One restriction is that a transaction boundary cannot span across an asynchronous boundary.

Figure 9. Sample assembly diagram

In Figure 9, there will be an asynchronous hop before the long running process component. This is because long running processes are implemented as asynchronous services. These services can only be invoked asynchronously. When they are invoked synchronously, such as the diagram shown in Figure 9, a Web service inbound call is synchronous, SCA will switch the call to asynchronous. The result of this switching introduces an asynchronous boundary, which means the transaction propagated from the caller will not reach the business process. You might notice that a long running process always has a JoinTransaction qualifier set to false for this reason.

Figure 10. Sample application with transaction boundary highlighted

In Figure 10, the dotted line surrounding LongRunningCreditCheck indicates that activities within the process may or may not run in the same unit of work. In addition to the component level transaction qualifiers discussed earlier, you can also configure transactional behavior attributes within a long running business processes. This enables you to influence transaction boundaries across multiple business activities.

Refer to the Transactional behavior of long-running processes topic in the WebSphere Process Server Information Center for information on how to configure this.
Advanced transaction qualifier usage

When a service component is wired to an asynchronous import binding, an optimization called hop reduction takes place (Figure 11). This optimization eliminates the inefficiency for a message to be put onto a queue to be picked up by the import, only to be put onto a queue again.

Figure 11. Asynchronous hop reduction impacts transaction boundary

Hop reduction is an optimization that happens automatically when the system detects that the import binding is asynchronous in nature. Specifically, JMS, MQ, and asynchronous SCA binding all have this optimized behavior. Because of hop reduction, it is possible to execute the import binding's put operation in the same unit of work that the calling component executes in.

Figure 12. Asynchronous import and transaction qualifier usage

Note that in Figure 12, the "Asynchronous Invocation" qualifier of C1 controls whether the put operation performed by AsynImport will be part of C1's unit of work. Again, the "joinTransaction" qualifier setting at both C1 and AsynImport are ignored due to the fact that both are invoked asynchronously.

In the current release (v6.2), grouping the put operation of an asynchronous import as part of the caller's unit of work is only possible if the asynchronous import is invoked asynchronously.

Deadlock problem

In a classic asynchronous request-response communication pattern, a deadlock occurs if the caller attempts to issue both the request sending and response retrieval in the same unit of work. The
thread is blocked waiting for a reply, whereas the real request is unable to commit to the outbound queue until the unit of work is completed.

The same problem occurs during asynchronous invocations of SCA services.

**Figure 13. Deadlock scenario**

```
public Object doSomething (Object arg) {
    Ticket t = s.invokeAsync(arg);
    Object r = s.invokeResponse(t, Service.WAIT);
    return r;
}
```

In the code snippet in Figure 13, `invokeResponse()` will be blocked indefinitely until a response has arrived. The request message is not committed to the outbound queue until the unit of work that `doSomething()` started is completed. You can get around this problem by setting the Asynchronous Invocation qualifier at the reference to `call`. This setting enables the `invokeAsync()` method to be executed in its own local transaction, which commits right away before the call is returned.

While a small configuration setting can get around the deadlock problem, the new setting introduces a new problem (Figure 14).

**Figure 14. Duplicate message scenario**

```
public Object doSomething (Object arg) {
    db.write(arg);
    Ticket t = s.invokeAsync(arg);
    // ... system crash!
    Object r = s.invokeResponse(t, Service.WAIT);
    return r;
}
```

Suppose the system crashes right after LTX1 has committed. TX1 will roll back. The system is left in an inconsistent state. A retry of TX1 later causes duplicated messages.

To avoid this situation, a deferred response usage conforms to one of the two scenarios discussed earlier in this article. Avoid the pattern of calling `invokeAsync()` and `invokeResponse()` in the same thread when a transaction is needed.

**Thread considerations**

Let's take a closer look at the threading implication of an asynchronous invocation.
A simple inter-component asynchronous invocation involves two to three threads. The caller that initiates the invocation puts the message onto a queue. That is one thread. A receiving thread is initiated to pick up the request message and dispatch to the target service. That is the second thread. If it is a request-response interaction, the service implementation would put the message onto a response queue. The response message is picked up by SCA and dispatched to the caller's callback method. This is the third thread.

So how many threads can you have in a module? Given that the module is deployed in an EAR container on WebSphere, the number of threads is drawn from the default thread pool configured at the server level as shown in Figure 15.

**Figure 15. Configuring thread pool size**

![Configuration page](image)

Notice in the sample interaction, two of the three threads were message driven bean (MDB) instances. In fact, the caller could have been initiated by a MDB if it was invoked asynchronously via an SCA Export. The number of concurrent SCA MDB threads in a module is defaulted to 10. This is controlled by the maximum concurrency level of the Activation Spec used by the MDB as illustrated in Figure 16.
If a single request-response asynchronous invocation requires at least three threads to complete, what happens to the system if there are thousands of messages entering the system to be processed? For applications that rely on asynchronous processing, adjusting the thread pool size and concurrency level of MDB instances based on your project's workload projection is almost mandatory.

For other concurrency tuning parameters, refer to IBM Redbook: WebSphere Business Process Management V6.1 Performance Tuning.

**Threading anti-pattern**

Occasionally adjusting MDB concurrency does not appear to help. In this case, review the application to make sure it does not contain any sync-over-async anti-pattern.
**Figure 17. Application with anti-pattern**

![Anti-pattern Diagram]

This anti-pattern (Figure 17) is characterized by wiring an asynchronous service to a synchronous export, or invoking an asynchronous service synchronously. When SCA converts the invocation from synchronous to asynchronous, the caller's thread is blocked waiting for a response to arrive. While the blocked threads consume valuable resources in the system, this can also lead to a deadlock situation where downstream components do not get any thread allocated to process the request because all the threads are consumed by the callers to wait for the response to return!

**Event sequencing**

Event sequencing guarantees related business events are delivered in the order they were sent. A corollary to the claim is that the guarantee is only applicable when data comes from the same source.

In a synchronous invocation, since a caller is blocked waiting for the response to return, it cannot issue a second request while waiting. Therefore, event sequences are automatically achieved between the source and the destination during synchronous invocations.

For asynchronous invocations, the caller can issue multiple requests without being blocked waiting for the responses. If the target service that handles the requests is multi-threaded, events can be processed out of sequence. The event sequencing qualifier is applied as an operational level qualifier for the target service so that processing sequence can be maintained.

Process Server event sequencing capability relies on the underlying messaging system to preserve messaging orders from the point where the caller places the message onto the request queue to the point where the message arrives to the final destination to be picked up for processing. Many messaging system offers this capability, so does WebSphere platform messaging, which Process Server uses for inter-component and inter-module asynchronous communications.

Since events can only go out of sequence during asynchronous invocations, the Event Sequencing qualifier is only needed for asynchronous service implementations or synchronous service that might be invoked asynchronously. In addition, special care needs to be taken for export bindings that are asynchronous in nature. Such bindings need to be executed in a single-threaded manner in order to preserve event order as shown in Figure 18.

**Figure 18. Configuring an application with event sequencing considerations**

![Event Sequencing Diagram]
The JMSExport in Figure 18 needs to be configured with a single MDB instance for processing incoming request. This is done by enabling the Event Sequencing attribute at the JMS Export. Due to hop reduction, MediationFlowComponent will run in the same thread as JMSExport, therefore the mediation component will naturally process events in sequence without the need for an event sequencing qualifier. LongRunningProcess will process events asynchronously, therefore the ES qualifier must be applied to the operations of LongRunningProcess to ensure the event order.

The ability to correctly choose an asynchronous boundary is essential in determining where to apply the Event Sequencing qualifier.

Error handling and recovery

There is a major difference between how errors are captured and managed between synchronous and asynchronous interactions.

System problems encountered by a service do not get returned to the caller if a service is invoked asynchronously. This behavior is generally applicable except when the caller is a long running BPEL process or a Mediation Flow Component.

So where does the failure go if it is not sent back to the caller? Most of these are captured by the Process Server Recovery subsystem. To understand further how errors are captured and routed in the system, see Exception handling in WebSphere Process Server and WebSphere Enterprise Service Bus.

Since the time the above article was published, enhancements have been made to the JMS binding such that failures occurred during processing by the JMS binding can also be managed via the Failed Event Manager. In addition, runtime problems encountered by BPC are also managed via the same tool (Figure 19).

Figure 19. Failed event management in Process Server 6.2
For more information about these enhancements, see the Managing failed events topic in the WebSphere Process Server Information Center.

When an error is captured in the Recovery subsystem, the user can resubmit the message from the Failed Event manager. Where is the point of re-injection? This depends on where the failed message was captured. This always takes place at asynchronous boundaries. Once again, the ability to discern asynchronous boundary in a given solution module is essential in understanding where messages are rolled back and where resubmit takes place.

A good understanding of Process Server recovery handling mechanism is only a starting point in building a robust system. Any production deployment has an error handling strategy in place to manage unexpected situations. The article Error handling in WebSphere Process Server, Part 1: Developing an error handling strategy provides a good starting point on this topic. Refer to the Related topics section at the end of this article for other reference material on this topic.

Network deployment

For high availability or workload scalability, you may have a need to cluster Process Server applications. There is a large amount of resources available that discuss how to configure Process Server in a network deployment environment. Most of them are listed in the Related topics section of this article.

Topology

All asynchronous messages processed by the WebSphere Platform Messaging system have to be routed through messaging engines. For Process Server, these asynchronous messages include internal SCA messages and JMS messages that happen to use WPM as the provider. The topic of how to place a messaging engine along side applications is of particular interest to Process Server users. Clustering WebSphere Process Server V6.0.2, Part 1: Understanding the topology by Michele Chilanti discusses different topologies with pros and cons of different topology tradeoffs.

Note that while the title of this article is for Process Server V6.0.2, the concepts described apply to any Process Server installations running on WebSphere version 6.x.

Message affinity

In his article, Chilanti illustrates a couple topologies with partitioned queues. You need to be aware that any topology involving partitioned queues is not suitable for Process Server asynchronous processing.

Process Server asynchronous communications often uses the deferred response pattern in its implementation. This pattern requires message affinity in a request-response interaction. Message affinity is the ability for a response message to be routed to the caller that issued the request. If cluster Member B issues a request to a remote service, the asynchronous response message needs to be received by cluster Member B, not cluster Member A. At the time of this article,
message affinity is only available on WebSphere Application Server v7.0. As a result, only deploy Process Server 6.2.x or earlier versions to a topology that does not use partitioned queues.

Event sequencing re-visit

If event sequencing is required, extra care must be taken when running applications in a clustered environment. The rule of thumb is to identify the entry point (for example, Export) and ensure messages can enter the system in a single-threaded fashion. If the entry point is a JMS or EIS Export that uses MDB to receive messages, the module containing the export binding is deployed on a single server. Once the message is received into the SCA module, you can dispatch it to another module deployed in a cluster using SCA binding.

Multiple messaging engines

Since Process Server asynchronous messages have to be routed through message engines, this begs the question of how to run multiple instances of this engine if it becomes a bottleneck. There are two choices to run multiple messaging engines:

- One way is to configure multiple active messaging engine instances in the same cluster. This act necessitates partitioned queues, which as discussed earlier, is not suitable for Process Server.
- Another way is to run the messaging engines in different clusters (for example, limit to one active messaging engine per cluster so to avoid partitioned queues). Each active messaging engine is responsible to host a different set of destinations.

In a multi-application cluster, multi-messaging engine cluster environment, you can connect an application to the "wrong" messaging engine when sending or receiving messages. That is, the messaging engine being connected might not be the one that hosts the target destination. This is a desirable behavior if the network has frequent glitches, for example. A message can be buffered by any running messaging engine (using remote queue points) until the hosting engine is available to process the messages. There are also scenarios where this behavior is undesirable. For example, event sequencing does not tolerate this type of buffering. The parameter to control whether Process Server connects to any available messaging engine or is required to connect to a particular engine at runtime is called target significance. As of Process Server 6.2, this parameter is scattered in multiple places in the system. It is best to apply the configuration using a script. Refer to the following article for where to download the script and how to configure this value if you are contemplating running more than one active messaging engine in your system: Configuring efficient messaging in multi-cluster WebSphere Process Server cells.

Conclusion

This article examined various topics related to asynchronous processing in Process Server:

- Clarified the concepts between asynchronous invocation and asynchronous services, and how they are related
- Reviewed different types of asynchronous invocations
• Categorized different asynchronous component types and binding types in Process Server and described how they interact with each other
• Asynchronous qualifiers, including expiration and transaction
• Threading considerations
• Event sequencing
• Error recovery
• Clustering considerations

While asynchronous processing has become an integral part of most complex business solutions today, you need to be aware of the complexities involved in design, development, and managing such systems. To design and deploy a robust asynchronous solution requires in-depth knowledge and careful planning. This article provides a glimpse into what is involved and pointers for further readings.
# Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code sample</td>
<td>samples.zip</td>
<td>136 KB</td>
</tr>
</tbody>
</table>
Related topics

• Programming model basics:
  • SCA service invocation API
  • SCA service implementation API
  • SCA asynchronous qualifiers
  • WebSphere Process Server invocation styles
• Transaction:
  • Transaction behavior of long running processes
  • Configuring transactions for mediation flow
• Error handling and recovery:
  • Exception handling in WebSphere Process Server and WebSphere Enterprise Service Bus
  • IBM Redbook: WebSphere Process Server Best Practices in Error Prevention Strategies and Solution Recovery
  • Error handling in WebSphere Process Server, Part 1: Developing an error handling strategy
• Performance:
  • IBM Redbook: WebSphere Business Process Management V6.1 Performance Tuning
• Scalability and high availability:
  • Clustering WebSphere Process Server V6.0.2, Part 1: Understanding the topology
  • Building clustered topologies in WebSphere Process Server V6.1
  • Configuring efficient messaging in multiclient WebSphere Process Server cells
  • Expanding clustered topologies for WebSphere Process Server and WebSphere Enterprise Service Bus