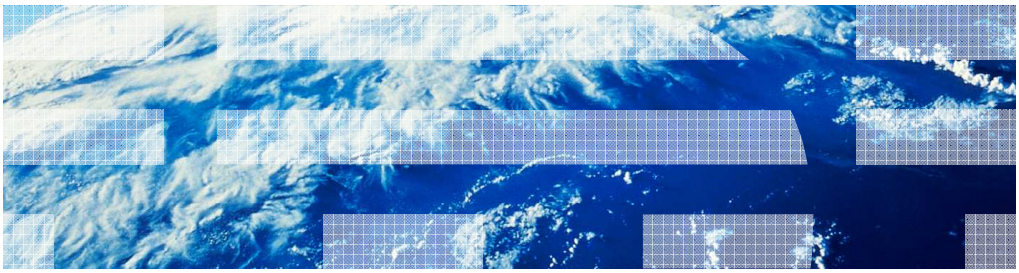
IBM WebSphere Application Server

Asynchronous invocation API



WebSphere software

This presentation describes the asynchronous invocation API, a new programming model for transferring events that require processing in a SIP application session to any server in a cluster based on an application session ID.

Agenda

- Overview
- Classes and interfaces
- Samples
- Use cases

The first section of this presentation provides an overview of the asynchronous invocation API, including a look at the purpose and benefits of the programming model and a description of the underlying architecture that the API uses. The second section describes the new classes that support the asynchronous invocation model, and the third section includes examples of how to use the APIs in your applications. The last section describes some use cases for the asynchronous invocation API, including both a single server and a clustered environment.

This section provides an overview of the asynchronous invocation API, including a look at the purpose and benefits of the programming model and a description of the underlying architecture that the API uses.

Overview

- Ideally, all events related to a particular SIP application session are:
  - Invoked on a single server initially assigned to handle that session
  - Processed on a single thread within that server
- In some situations, an external event might require transferring work from one server in a cluster to another
  - Typically triggered by an event outside the SIP container
  - Causes a state change in a SIP session on another server in the cluster
- Asynchronous invocation API allows an application to send a task to be run on the correct server
  - Dispatched to the appropriate location, based on session ID

Typically, all events related to a certain sip application session should be invoked on one server initially assigned to handle that session and on a single thread within that server. There are some situations when an event other then a SIP container event – not the reception of a SIP message or SIP timer execution – is invoked on a certain server in the cluster that it needs to cause a state change to a SIP session that resides on another server. In this case, the application needs to pass the event task to be processed on the correct server. The idea behind an asynchronous invocation solution is that a piece of work associated with a specific session can be handled by any server in the cluster, and this server can transfer the work directly to the right place, based on the session ID. The actual work is then done in the target server where the SIP application session is managed. The most ideal case is when the application session resides on the server that receives the message related to the session and this server performs the task locally.

Problems the API solves

- Synchronicity is now guaranteed when requests for interaction with the SIP container come from other Java EE components
  - Previous implementations handled SIP servlet requests only, for requests within the server that owned the application session
- Two requests related to the same SIP application session cannot run simultaneously on different threads
  - **Example:** An MDB in a Java EE application retrieves an event to send a SIP message on a thread that is different than the thread that owns the SIP application session
- In the context of a SIP application session, a server can redirect both SIP and non-SIP requests to the owning server
  - **Example:** A web service that is initiating a SIP dialog can reside in a server different than the server that owns the SIP application session it needs to use

The asynchronous invocation model guarantees that no two tasks dispatched using the API, associated with the same session ID, can be processed at the same time on different threads, even when those tasks come from non-SIP interfaces, like Java EE components. Previous implementations handled SIP servlet requests only, and can only process the task within the same server that owned the application session – there was no native support for dispatching across a cluster. Consider a message-driven bean in a Java EE application that retrieves an event that requires sending a SIP message on a particular thread, associated with some specific SIP application session. If the SIP message is dispatched using the asynchronous invocation API, then no other tasks associated with that session ID are allowed to run with the asynchronous event is being processed. Synchronicity is guaranteed even when SIP messages are dispatched across servers in a cluster. For example, a web service that is initiating a SIP dialog can reside in a server that is different than the server that owns the SIP application session that the web service needs to use.

Benefits

- Avoids the need to access a central session repository for all servers and migrate sessions from one server to another to work with remote sessions
- Works in a thread-safe manner
  - API guarantees that only one thread at a time processes messages related to a SIP application session, so no need to synchronize session access
  - Synchronicity guaranteed in both a single server and a clustered environment
- No more than two servers are involved in the invocation process
  - The server that retrieves the work task
  - The target server that owns the SIP application session relevant to the task
- Asynchronous invocation API is scalable
  - Cross server invocation is used only when required
  - No impact to performance in a clustered environment as more servers are added to the cluster

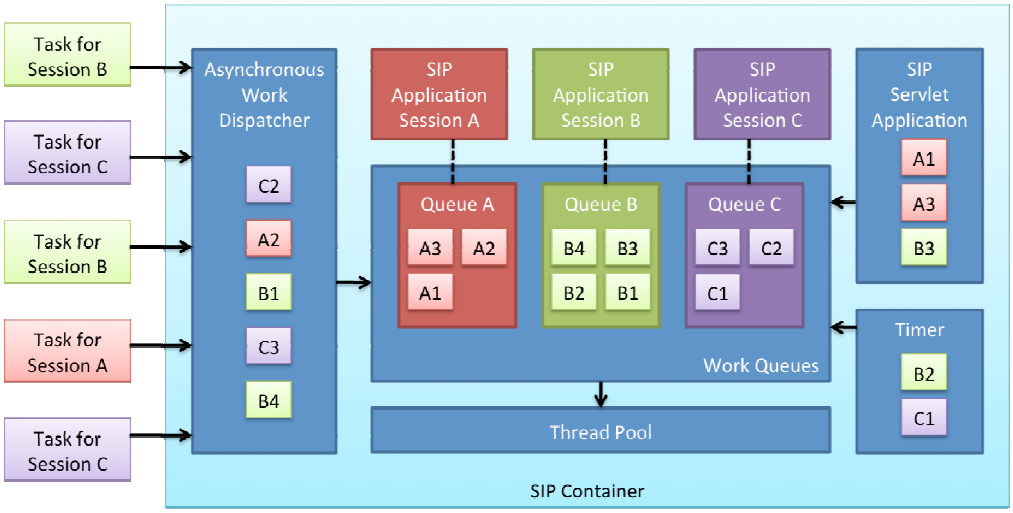
By using the asynchronous invocation API, you avoid the need to maintain and access a central session repository for all servers, and you do not need to migrate sessions from one server to another to work with remote sessions. This is good because using those methods impacts performance due to locking, maintaining process synchronicity, and remotely accessing data. The asynchronous invocation API guarantees synchronicity and processes all dispatched tasks in a thread-safe way. The API guarantees that only one thread at a time processes messages related to a SIP application session, so you do not need to synchronize session access. Synchronicity is guaranteed in both a single server and a clustered environment. Using the API offers much better performance than a manual locking scheme, and the API is scalable since no more than two servers are ever involved in the invocation process – the server that invokes the task, and the target server that owns the SIP application session relevant to the task. Cross server invocation is used only when required, and, in a clustered environment, there is impact to performance as more servers are added to the cluster.

Asynchronous task dispatching

- Application session is created by an incoming request, or dynamically by an application
- SIP container has queues where tasks related to SIP application sessions wait to be processed
  - Tasks are pieces of application code that process SIP messages or modify objects associated with a particular session
  - Examples: Listener timeout or processing a SIP message
- When a new task is in the queue, one of the threads in the thread pool will process it
  - All tasks related to a given session ID are processed by an appropriate thread
  - Avoids synchronicity and locking issues

A SIP application session can be created in different ways – for example, by an incoming SIP request, or dynamically by some application. Once a SIP application session is created, all related SIP messages always get delivered to the same server in the cluster; this ensures that sessions reside in the same SIP container. The container has queues where tasks related to a SIP application session waits to be processed. These queues are not exposed through any interfaces or APIs, instead, all events targeted to the same session are always automatically sent to the same queue internally. These tasks can be any piece of application code that processes a SIP message or modifies objects associated with a particular session. When a task comes into the queue, one of the threads in the thread pool processes that task. No two tasks from the same queue are dispatched to the thread pool at the same time. It is not true that all tasks for a particular session are processed on the same thread in the thread pool – only that they will come in through the same queue and therefore not be processed by different threads at the same time. By using this queuing scheme, the asynchronous invocation API avoids synchronicity and locking issues.

Asynchronous task dispatching, single server



This diagram illustrates how asynchronous task dispatching works inside the SIP container. There are three SIP application sessions – A, B, and C. Each one is associated with a specific queue. All events targeted to a specific session get added to the appropriate queue and are processed in the order received. These events can come from many different locations, including SIP servlet applications, SIP timers, Java EE applications, or tasks being dispatched using the asynchronous invocation API. Consider SIP application session A in this example. There are tasks coming in for session A both from the SIP servlet application, shown on the right, and the asynchronous work dispatcher, shown on the left. As each event targeted for session A comes in to the container, it automatically gets added to queue A, which is designated to handle all of the tasks related to session A. Only one item from queue A can be sent to the thread pool for processing at any given time. In this way, it is guaranteed that no two events related to the same application session can be processed at the same time.

Section

Classes and interfaces

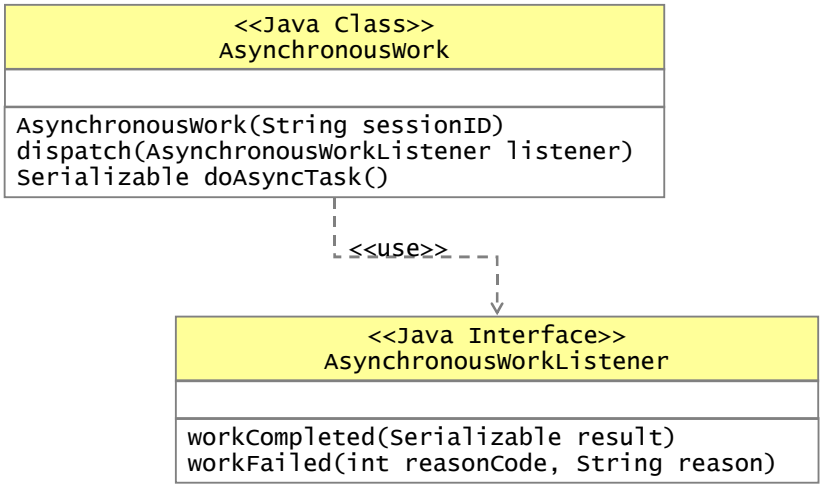
This section describes the new classes that support the asynchronous invocation model.

API packaging

- The classes and interfaces for the asynchronous invocation API are packaged here:
  - <WAS_HOME>/plug-ins/com.ibm.ws.sip.interface.jar
- One class and one interface are included as a part of the asynchronous invocation API:
  - **AsynchronousWork:** class used by an application developer to run asynchronous work task
  - **AsynchronousWorkListener:** interface used by an application developer to receive a response for the result of the asynchronous work task

The classes and interfaces that are required to implement the asynchronous invocation model are provided with WebSphere Application Server. They are packaged in the <WAS_HOME>/plug-ins directory in the file com.ibm.ws.sip.interface.jar. The API is simple to use; there is only one class and one interface that you need to add asynchronous dispatching to your application. The AsynchronousWork class is used to run the asynchronous work task. The AsynchronousWorkListener interface receives responses – for example, a success or failure response – for an asynchronous task.

Asynchronous invocation class diagram



This class diagram shows how the AsynchronousWork class and the AsynchronousWorkListener are related. The AsynchronousWork class contains three important methods: a constructor that takes a session ID as a parameter, a dispatch method that sends the asynchronous task off to be processed and takes a listener as a parameter, and a doAsyncTask method that defines the piece of work that gets dispatched and runs in the target session's queue. The AsynchronousWorkListener includes two methods that get signaled on success or failure of the task dispatch. If the task is dispatched successfully, a Serializable result comes back to the originating thread. If the dispatch fails, a description of what failed comes back through the workFailed method.

AsynchronousWorkListener

- Interface used by an application developer to receive the result of an asynchronous work task
  - Either success or failure
- Contains two methods for processing the result
  - void workCompleted(Serializable result)
  - void workFailed(int reasonCode, String reason)
    - Called by the container when the dispatch fails, not when the task fails

The AsynchronousWorkListener interface can be used by an application developer to define a listener that is able to receive the result of an asynchronous work task. This result can take two forms – either success, or failure. If the task dispatch is successful, a Serializable result object comes back through the listener. Just because the workCompleted method gets signaled, that does not mean that the asynchronous piece of work completed with the expected results. It is a good idea to verify the result of the task before proceeding with application processing. If the task dispatch fails, then the workFailed method is called by the container. Note that if the workFailed method gets invoked, that does not mean that the task failed, only that the dispatch of the task failed. One reason that the workFailed method might be called is if the SipApplicationSession was not found. This can happen if the session with the ID that was used never existed, or was already invalidated and removed before the task was dispatched.

When the asynchronous task is dispatched from a non-SIP container thread – for example, a JMS thread – the listener callback code is run on the same SIP thread that the asynchronous task was dispatched to. Alternatively, when the asynchronous task is dispatched from a SIP container thread – for example, to modify a SipApplicationSession element other than the session that the dispatching thread is currently set to handle – then the listener callback is dispatched back to be run in the same queue where the task that issued the dispatch was taken from. This is so that the application can continue modifying the first application session data after the dispatch for the second session has taken place, and is able to do that in the context of the right thread. When the dispatch starts from a non-SIP container thread, there is no way to return it back to the originating thread.

AsynchronousWork (1 of 2)

- Abstract class that needs to be extended by the class wanting to do asynchronous work
- Extending class must implement the abstract method doAsyncTask() that returns a Serializable object
  - The Serializable that gets returned is the one that is passed to the AsynchronousWorkListener.workCompleted() method
- When an application wants to do asynchronous work:
  - Provides a session ID to create an AsynchronousWork object
  - Calls the dispatch() method with an optional AsynchronousWorkListener
    - The listener receives a response when the work is complete
    - If the calling application does not want a response, pass in null for the listener

AsynchronousWork is an abstract class that needs to be extended by the class wanting to dispatch asynchronous tasks. The extending class must implement the doAsyncTask method that returns a Serializable object. This Serializable object is the same one that gets returned to the AsynchronousWorkListener when the task completes. If null comes back from the doAsyncTask method, the listener does not get invoked, even if it exists. When an application wants to dispatch an asynchronous task, it creates an AsynchronousWork object based on a SIP application session ID. This AsynchronousWork object must contain a doAsyncTask method that defines the piece of work that is going to be invoked asynchronously. When the application wants to invoke the task, it calls the dispatch method on the AsynchronousWork object, passing in an optional AsynchronousWorkListener to receive a response when the work is complete. If the calling application does not want a response, it can pass in null for the listener.

AsynchronousWork (2 of 2)

- When the container receives the dispatch() call, it:
  - Determines which server manages the related session
  - Invokes the doAsyncTask() method in the correct server, on an appropriate thread
- After the work is done, a response comes back to the container
  - Invokes the appropriate AsynchronousWorkListener to return a response to the original calling application

When the SIP container receives a dispatch call from an application, it needs to determine which server manages the related session. If the session is managed in a different server, the piece of work is transferred to the owning server. The task is sent to the queue associated with the owning session ID, so it will run on an appropriate thread, avoiding any conflicts with other tasks targeted for that session ID. After the work is done, a response comes back to the container, and the container invokes the appropriate AsynchronousWorkListener to return a response to the original calling application. If the work was transferred across servers in a cluster, the response gets transferred back to the calling server, to the original thread where the asynchronous task was invoked.

Section

Samples

This section gives examples of how to use the APIs in your applications.

Setting up the listener

```
public class MyListener
implements AsynchronousWorkListener {

    public void workCompleted(Serializable myResult){

    // perform action based on successful work
    }

    public void workFailed(int reasonCode, String reason){

    // perform action based on failed work
    }
}
```


To receive information about the task completion, implement the AsynchronousWorkListener class, as in the example shown here. This interface defines two methods – workCompleted and workFailed – to receive information about the result of the asynchronously dispatched task. It is a good practice to verify that the completion result is valid before proceeding with application processing. The code in these methods is invoked on the source server.

Creating the class to do asynchronous work (1 of 2)

```
public class MyClass extends AsynchronousWork {
    ...

    // This is the code that is invoked on the
    // target machine/thread
    public Serializable doAsyncTask() {
        // Application code goes here, for instance
        appSession =
          sessionUtils.getApplicationSession(_sessionId);
        appSession.createRequest();
        Serializable myResponse = new MyResponse();
        myResponse.setStatus(200);
        return (myResponse);
    }
    ...
```


Extend the abstract class AsynchronousWork with the SIP related code that you need to dispatch to the appropriate server, based on session ID. The extended implementation of the doAsyncTask method is invoked on the target server that contains the SIP application session, and whose ID was set in the constructor that implements the AsynchronousWork class. The doAsyncTask method can contain any application code that you want to run on the target server for the asynchronous invocation.

Creating the class to do asynchronous work (2 of 2)

```
public class MyClass extends AsynchronousWork {
    ...
    // Making the call to do asynchronous work
    public void onMyMessage() {
        // Obtain the session ID from the message or
        // somewhere else
        String sessionId = obtainIdFromMessage();
        MyClass myClass = new MyClass(sessionId);

        // Create the listener
        MyListener myListener = new MyListener();

        // Dispatch it
        myClass.dispatch(myListener);
     }
}
```


This page offers an example of how to actually make the asynchronous call. In this case, the onMyMessage method is an application method that gets called when some proprietary message is received – the trigger to dispatch an asynchronous piece of work will vary from application to application. Recall that, in this example, the MyClass object extends AsynchronousWork, so you need to provide a session ID to create an instance of MyClass. In this case, the incoming message that's triggering the asynchronous task contains the session ID that you need, so this application contains a private method called obtainIdFromMessage that will pull out the required session ID (the implementation of the private method is not shown in this example). In your application environment, you might obtain the session ID using some other mechanism – perhaps it is stored in a registry that you maintain, or perhaps the session ID will come in as a part of a web service message that is requesting some SIP action. After acquiring the session ID, create an instance of the class based on AsynchronousWork (in this case, MyClass), then create an instance of your AsynchronousWorkListener, if you are using one. Finally, invoke the dispatch() method to send the asynchronous task off for processing. The result of the task comes back through the listener.

Section

Use cases

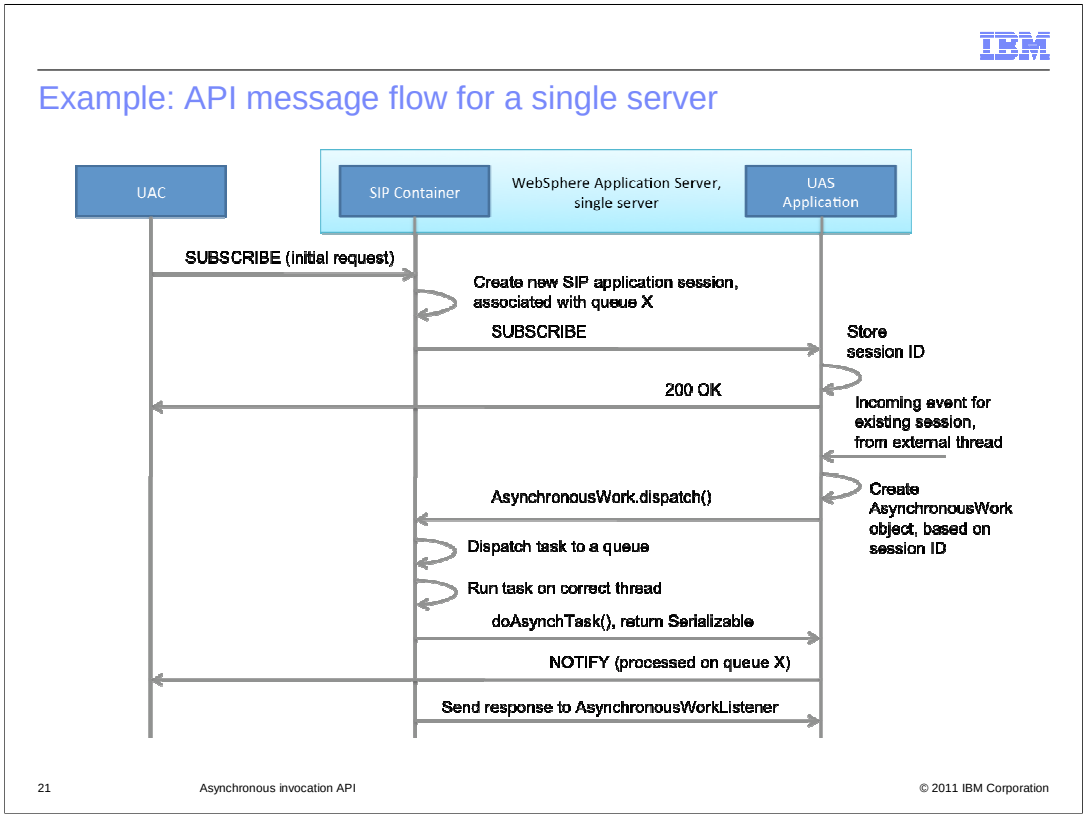
This section describes some use cases for the asynchronous invocation API, including both a single server and a clustered environment.

Use case examples

- Java EE application receives an event to send a message to an application session that does not belong to the SIP container
- Application sends an instant message on an existing SIP dialog using the HTML protocol
  - Instant message handled by a web service that resides on a server that does not own the dialog
  - Message contains the application session ID for which the instant message was generated
    - Allows the SIP container to pass the message to the correct server

Several compelling use cases for the asynchronous involve interaction with non-SIP components. For example, a Java EE application might receive an event – perhaps through a JMS message – that it needs to send a message to an application session that does not belong to the SIP container in the server where the Java EE application is running. In this case, the application can use the asynchronous invocation API to dispatch the message to the correct server, in a thread-safe way.

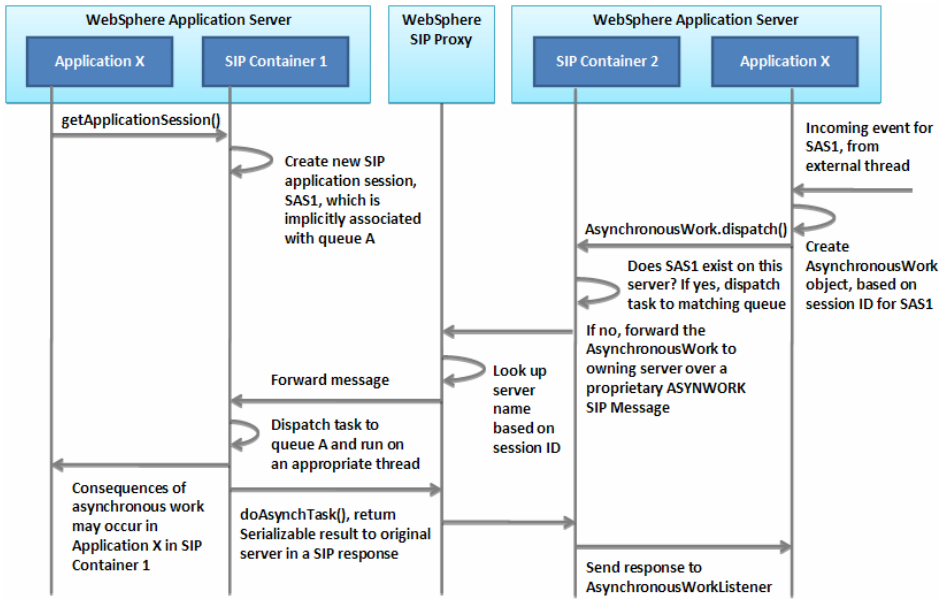
Consider an application that combines instant messaging and web services with SIP messages. The application needs to send an instant message using the HTML protocol that needs to be handled in an existing SIP dialog. The instant message is handled by a Web service that resides on a server that does not own the dialog, but the message contains the SIP application session ID for which the instant message was generated. This allows the SIP container to pass the message to the correct server, using the asynchronous invocation API.



This diagram shows an example of the message flow for an application using the asynchronous invocation API, in a single server environment, so the application making the asynchronous calls is running in the same instance of WebSphere Application Server as the SIP container that owns the relevant SIP application session.

Starting on the left of the diagram, a client, or UAC, sends a SUBSCRIBE message to the SIP container, indicating a need to be notified of certain types of messages. This initial request causes a new SIP application session to be created; this session is internally associated with a processing queue – in this case, queue X. The SUBSCRIBE request gets forwarded to the UAS application, which is running in the same server. The UAS stores the session ID for later use, and sends a 200 OK back to the UAC. At some point in the future, the UAS receives an incoming event for the existing SIP application session associated with the UAC. This event is coming from an external thread – perhaps on a JMS queue or a web service notification. This event is shown by the incoming arrow on the far right of the diagram. When the UAS receives this event, it gets ready to invoke the work asynchronously by creating an AsynchronousWork object based on the session ID that came in as a part of the incoming message. The UAS calls the dispatch() method, and the SIP container sends the task to the queue for the correct application session, and the task gets run on an appropriate thread for that session's queue. In this case, the NOTIFY message is being sent as a part of the asynchronous task. When the task completes, the SIP container sends status back to the UAS application through the AsynchronousWorkListener.

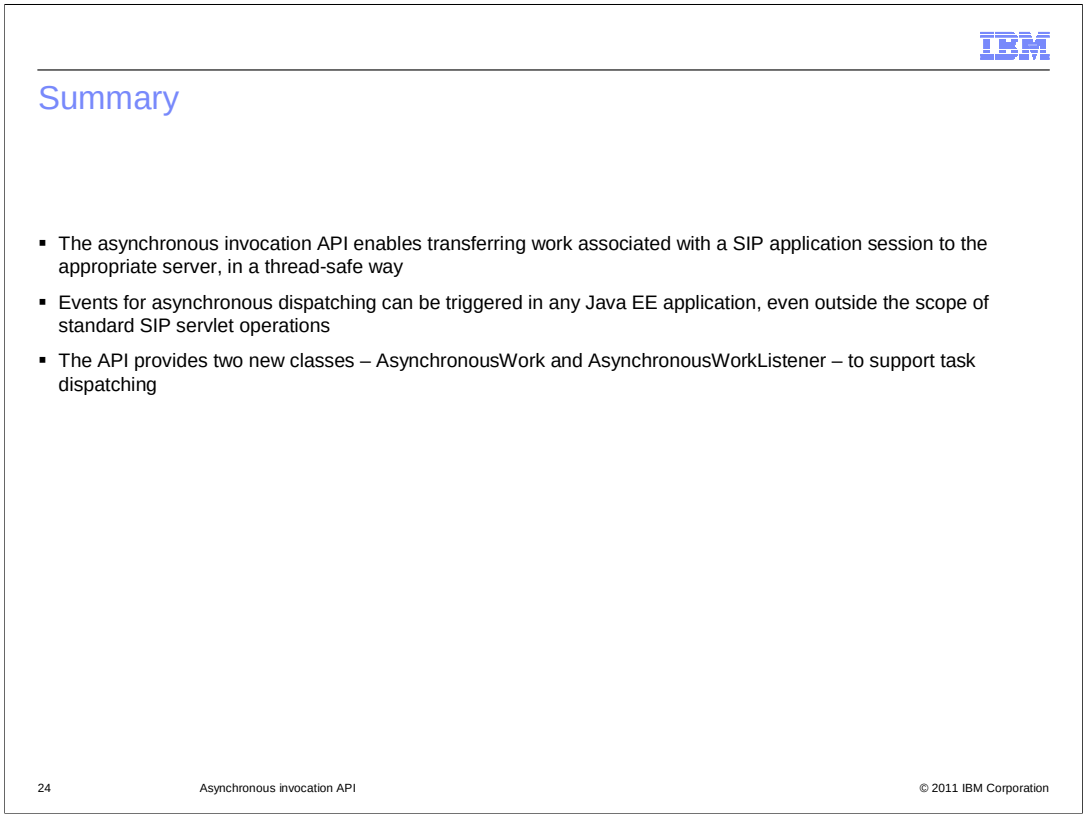
Example: API message flow in a cluster



This diagram shows an example of asynchronous invocation API message flow when the invocation takes place across servers in a cluster.

In this case, Application X is running on SIP Container 1, and is associated with a SIP application session, SAS1. This relationship is shown in the upper left of the diagram. The session SAS1 is implicitly associated with a specific internal queue that will process all of the messages targeted for that session – queue A in this example. The same application is running on SIP Container 2 in a separate instance of WebSphere Application Server. At some point, Application 2 receives an incoming event targeted for SAS1, as shown by the arrow coming in from the far left of the diagram. Application 2 uses the asynchronous invocation API to dispatch this event by creating an AsynchronousWork object based on the session ID SAS1, and then dispatching the task. SIP Container 2 receives this dispatched task and checks whether SAS1 exists on this server. In this case it does not, and SIP Container 2 sends the task over a SIP ASYNWORK message to the WAS SIP Proxy. The Proxy in turn looks up the appropriate server name based on the session ID, and then forwards the SIP request with the asynchronous task to the owning server. From this point on, the work takes place much like it does in a single server context – the asynchronous work time is added to the processing queue for SAS1, the task is run on an appropriate thread, and the result of the work is transferred back to the originating server, using a SIP response, through the SIP proxy.

This section contains a summary of this presentation.



The asynchronous invocation API enables transferring work associated with a SIP application session to the appropriate server, in a thread-safe way. Task dispatching can take place across servers in a cluster. Events for asynchronous dispatching can be triggered in any Java EE application, even outside the scope of standard SIP servlet operations. For example, events targeted for a SIP dialog might come in on a JMS queue or through a web service or an enterprise bean, and the events can be dispatched to be processed on the correct target server that owns the relevant SIP application session. The API includes a new class, AsynchronousWork, and a new interface, AsynchronousWorkListener, to support task dispatching.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WASv8_SIP_AsyncInvocationAPI.ppt

This module is also available in PDF format at: ../WASv8_SIP_AsyncInvocationAPI.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information