Platform SDK Developer's Guide

Event Handling

4/9/2025

Event Handling

Java

Once you have connected to a server, much of the work for your application involves sending messages to that server and handling the events you receive from the server. This article describes how to send and receive messages from a server.

Messages: Overview of Events and Requests

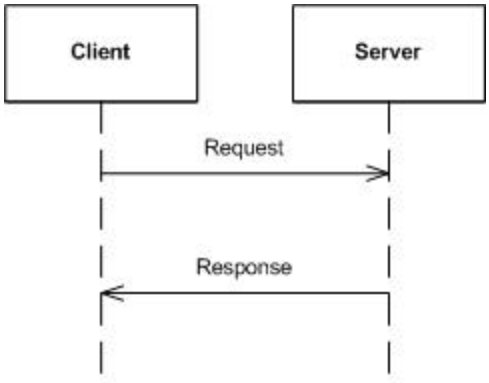
Messages you send to a server are called requests, while messages you receive are called events. An event that is received from a server as the result of executing a request is called a response. In summary, messages can be classified by using the following taxonomy:

- Requests: sent to the server
- Events: received from the server
  - Responses: received as the result of a request
  - Unsolicited events: not a direct result of a request

> **Tip**
> On this page, we will use the more general term "message" instead of "event", in order to avoid confusion between protocol events and programming events.

For example, you may send a request to log in an agent or to gather statistics. You might also send a request to update a configuration object, or to shut down an application.

In each of these cases, the server will respond with an event message, as shown below.



You may also get unsolicited events from the server. That means receiving events that are not a response to a specific request. For example, `EventRinging` will notify you of a call ringing on an extension that you are currently monitoring.

Receiving Messages

With the Platform SDK, you can receive messages synchronously or asynchronously. It is important that you define the way your application will work in this aspect. In general, you will probably use only one type or the other in the same application.

Interactive applications normally use asynchronous message handling, because that will prevent the UI thread from being blocked, which could make the application appear "frozen" to a user. On the other hand, non-interactive batch applications commonly use synchronous response handling, as that allows writing easy code that performs step-by-step.

Receiving Messages Asynchronously

Most Platform SDK applications need to handle unsolicited events. This is particularly true for applications that monitor the status of contact center resources, such as extensions.

You receive server messages by implementing a `MessageHandler` that contains the event-handling logic:

```
[Java]

MessageHandler tserverMessageHandler = new MessageHandler() {
        @Override
        public void onMessage(Message message) {
                // your event-handling code goes here
        }
};
```

Then you set your implementation as the protocol `MessageHandler`.

```
[Java]

tserverProtocol.setMessageHandler(tserverMessageHandler);
```

> **Important**
>
> You need to know that your event-handling logic will be executed by using the protocol invoker. Please set the invoker appropriate for your application needs. For more information about the protocol invoker and how to set it, refer to Connecting to a Server.

Inside your event-handling code, you will want to execute different logic for different kinds of events. A typical way to do this is using a `switch` statement, based on the event identifier:

```
[Java]

switch (message.messageId()) {
        case EventAgentLogin.ID:
                OnEventAgentLogin(message);
                break;
        case EventAgentLogout.ID:
                OnEventAgentLogout(message);
                break;
}
```

Receiving Messages Synchronously

Some kinds of applications, such as batch applications, benefit from receiving messages synchronously. This means that received messages will queue up and be handled by the application on demand.

In order to receive messages this way, you simply **do not** set a protocol `MessageHandler` as described in the previous section.

> **Tip**
>
> For releases prior to Platform SDK 8.1.1, messages were received synchronously by default. Please note that 8.1.1 behavior is backwards-compatible, and pre-8.1.1 applications will continue to work as expected without any modification.

To receive a message synchronously, use the `Receive` method. This method blocks processing, waiting for the next message to be received before continuing. Take into account that the maximum time to wait is set by a configurable timeout value. If the timeout expires and no event is received, you will receive a `null` value.

```
[Java]

Message message = tserverProtocol.receive();
```

If you want to set your own timeout, you can use the `Receive` method overload that takes a timeout parameter. Otherwise, if you use `Receive` with no parameters, the protocol Timeout property will be used.

Sending Requests Asynchronously

This is the easiest way to send a message to a server. Suppose you have created and filled a request object, for example, a `RequestAgentLogin` message for Interaction Server:

```
[Java]

RequestAgentLogin loginRequest = RequestAgentLogin.create();
loginRequest.setTenantId(tenantId);
loginRequest.setAgentId(agentId);
loginRequest.setPlaceId(placeId);
```

You can then send it to the server using the following code:

```
[Java]

interactionServerProtocol.send(loginRequest);
```

This will result in your application receiving a response from the Interaction Server: either an `EventAck` or an `EventError` message. By using the Send method, you will ignore that response at the place where you make the request. You will get the response, like any other unsolicited event, using the techniques described in the *Receiving Messages* section.

Handling Responses

The understanding of how to send requests and receive events is all you need to communicate with Genesys servers. However, the Platform SDK also provides the ability to easily associate a response with the particular request that originated it.

Receiving a Response Synchronously

The easiest way to handle responses is with the `Request` method. This is a blocking method, as your application stops to wait for a response to come from the server. Using the same request example above:

```
[Java]

Message response = interactionServerProtocol.request(loginRequest);
if (response.messageId() == EventAck.ID) {
        EventAck eventAck = (EventAck)response;
        // continue here
}
else {
        // handle the error here
}
```

Notice that you will need to cast the message to a specific message type in order to access its attributes. If a request fails on the server side, you will typically receive an `EventError`.

Take into account that the `Request` method blocks until a message is received or a timeout occurs. If the timeout elapses and no response was received from the server, then a `null` value is received. The timeout parameter can be specified in the request method. If you do not use the timeout parameter then, then the protocol `Timeout` property is used.

The Request method will only return one message from the server. In the case that the server returns subsequent messages, apart from the first response, as a result of the requested operation, then you must process those messages separately as unsolicited events. Please make sure that your code handles all messages received from your servers.

When using the Request method, your application only receives the response to that request as a return value. The response will not be received as an unsolicited event as well. (You can change this behavior by using the CopyResponse protocol property, described below.)

Receiving a Response Asynchronously

For many applications, blocking your thread while waiting for a response to your request is not appropriate. For example GUI applications, where the GUI can appear "frozen" if the response takes too much time to be received. It can also be true for batch applications that may want to send multiple requests at the same time, while waiting for all responses concurrently. For these scenarios it is possible to receive responses asynchronously.

Receiving a Response Asynchronously Using a Callback

By using requestAsync, your thread will not block, and it will permit you to handle the response by using callback methods that will get called asynchronously.

First, you will need to implement a CompletionHandler which will contain the logic for handling the response to your request:

```
[Java]

private static final CompletionHandler loginResponseHandler = new  CompletionHandler() {

        @Override
        public void completed(Message message, Void notUsed) {
                // handle message here
        }

        @Override
        public void failed(Throwable exc, Void notUsed) {
                // handle error here
        }

};
```

> **Important**
>
> The CompletionHandler callback methods will be executed by the protocol invoker.

Then you can use the CompletionHandler as a parameter to the requestAsync method:

```
[Java]

interactionServerProtocol.requestAsync(loginRequest, null, loginResponseHandler);
```

Notice that in this example, the attachment parameter has not been used. If you are sharing the same CompletionHandler implementation for handling the responses to different requests then you

may want to use an attachment to make it easy to differentiate among those requests.

Receiving the Response as a Future

Alternatively, you may want to handle responses using the same thread that did the request, but have the option to do something concurrently while waiting for the response. To accomplish this, use the `beginRequest` method.

As an example, you might perform two agent login requests concurrently: one for logging into the T-Server, and another for logging into Interaction Server.

```
[Java]

RequestFuture loginVoiceFuture = tserverProtocol.beginRequest(loginVoiceRequest);
RequestFuture loginMultimediaFuture =
interactionServerProtocol.beginRequest(loginMultimediaRequest);

Message loginVoiceResponse = loginVoiceFuture.get();
Message loginMultimediaResponse = loginMultimediaFuture.get();

// handle responses, both are available now
```

When using the `requestAsync` or `beginRequest` methods, you will **not** receive the response as an unsolicited event. (You can change this behavior by using the CopyResponse protocol property, described below).

CopyResponse

Previously it was stated that responses returned by request methods are not received as unsolicited events by default. This behavior can be modified by using the protocol `CopyResponse` property. The default value is `false`, but it can be set to `true` like this:

```
[Java]

tserverProtocol.setCopyResponse(true);
```

This is particularly useful for protocols which define events that can be both received unsolicited and as a response to a client request (such as `EventAgentLogin` defined by the T-Server protocol). By setting the `CopyResponse` property to `true`, you can execute your agent state change logic only when handling the message as an unsolicited event, and you do not need to include it when receiving the message as a response.

.NET

Once you have connected to a server, much of the work for your application will involves sending messages to that server and handling the events you receive from the server. This article describes how to send and receive messages from a server.

Messages: Overview of Events and Requests

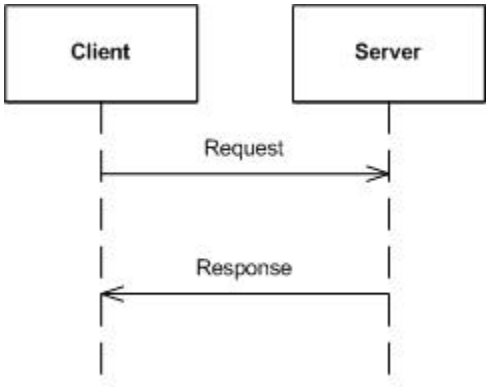
Messages you send to a server are called requests, while messages you receive are called events. An event that is received from a server as the result of executing a request is called a response. In summary, messages can be classified by using the following taxonomy:

- Requests: sent to the server
- Events: received from the server
  - Responses: received as the result of a request
  - Unsolicited events: not a direct result of a request

> **Tip**
> On this page, we will use the more general term "message" instead of "event", in order to avoid confusion between protocol events and programming events.

For example, you may send a request to log in an agent or to gather statistics. You might also send a request to update a configuration object, or to shut down an application.

In each of these cases, the server will respond with an event message, as shown below.



You may also get unsolicited events from the server. That means receiving events that are not a response to a specific request. For example, `EventRinging` will notify you of a call ringing on an extension that you are currently monitoring.

Receiving Messages

With the Platform SDK, you can receive messages synchronously or asynchronously. It is important that you define the way your application will work in this aspect. In general, you will probably use only one type or the other in the same application.

Interactive applications normally use asynchronous message handling, because that will prevent the UI thread from being blocked, which could make the application appear "frozen" to a user. On the other hand, non-interactive batch applications commonly use synchronous response handling, as that allows writing easy code that performs step-by-step.

Receiving Messages Asynchronously

Most Platform SDK applications need to handle unsolicited events. This is particularly true for applications that monitor the status of contact center resources, such as extensions.

You receive server messages asynchronously by subscribing to the `Received` .NET event:

```
[C#]

tserverProtocol.Received += OnTServerMessageReceived;
```

Then you can implement your event-handling logic:

```
[C#]

void OnTServerMessageReceived(object sender, EventArgs e)
{
        IMessage message = ((MessageEventArgs)e).Message;
        // your event-handling code goes here
}
```

> ### Important
> You need to know that your event-handling logic will be executed by using the protocol invoker. Please set the invoker appropriate for your application needs. For more information about the protocol invoker and how to set it, refer to Connecting to a Server.

Inside your event-handling code, you will want to execute different logic for different kinds of events. A typical way to do this is using a `switch` statement, based on the event identifier:

```
[C#]

switch (message.Id)
{
        case EventAgentLogin.MessageId:
                OnEventAgentLogin(message);
                break;
        case EventAgentLogout.MessageId:
                OnEventAgentLogout(message);
                break;
}
```

Receiving Messages Synchronously

Some kinds of applications, such as batch applications, benefit from receiving messages synchronously. This means that received messages will queue up and be handled by the application on demand.

In order to receive messages this way, you simply do not subscribe to the `Received` .NET event as described in the previous section.

> **Tip**
>
> For releases prior to Platform SDK 8.1.1, messages were received synchronously by default. Please note that 8.1.1 behavior is backwards-compatible, and pre-8.1.1 applications will continue to work as expected without any modification.

To receive a message synchronously, use the `Receive` method. This method blocks processing, waiting for the next message to be received before continuing. Take into account that the maximum time to wait is set by a configurable timeout value. If the timeout expires and no event is received, you will receive a `null` value.

```
[C#]

IMessage message = tserverProtocol.Receive();
```

If you want to set your own timeout, you can use the `Receive` method overload that takes a timeout parameter. Otherwise, if you use Receive with no parameters, the protocol `Timeout` property will be used.

Sending Requests Asynchronously

This is the easiest way to send a message to a server. Suppose you have created and filled a request object, for example, a `RequestAgentLogin` message for Interaction Server:

```
[C#]

var loginRequest = RequestAgentLogin.Create();
loginRequest.TenantId = tenantId;
loginRequest.AgentId = agentId;
loginRequest.PlaceId = placeId;
```

Then you can send it to the server:

```
[C#]

interactionServerProtocol.Send(loginRequest);
```

This will result in your application receiving a response from the Interaction Server: either an `EventAck` or an `EventError` message. By using the Send method, you will ignore that response at the place where you make the request. You will get the response, like any other unsolicited event, using the techniques described in the *Receiving Messages* section.

Handling Responses

The understanding of how to send requests and receive events is all you need to communicate with Genesys servers. However, the Platform SDK also provides the ability to easily associate a response with the particular request that originated it.

Receiving a Response Synchronously

The easiest way to handle responses is with the `Request` method. This is a blocking method, as your application stops to wait for a response to come from the server. Using the same request example above:

```
[C#]

IMessage response = interactionServerProtocol.Request(loginRequest);
if (response.Id == EventAck.MessageId)
{
        var eventAck = (EventAck)response;
        // continue here
}
else
{
        // handle the error here
}
```

Notice that you will need to cast the message to a specific message type in order to access its attributes. If a request fails on the server side, you will typically receive an `EventError`.

Take into account that the `Request` method blocks until a message is received or a timeout occurs. If the timeout elapses and no response was received from the server, then a `null` value is received. The timeout parameter can be specified in the request method. If you do not use the `timeout` parameter then the protocol `Timeout` property is used.

The `request` method will only return one message from the server. In the case that the server returns subsequent messages, apart from the first response, as a result of the requested operation, then you must process those messages separately as unsolicited events. Please make sure that your code handles all messages received from your servers.

When using the `Request` method, your application only receives the response to that request as a return value. The response will not be received as an unsolicited event as well. (You can change this behavior by using the `CopyResponse` protocol property, described below).

Receiving a Response Asynchronously

For many applications, blocking your thread while waiting for a response to your request is not appropriate. For example GUI applications, where the GUI can appear "frozen" if the response takes too much time to be received. It can also be true for batch applications that may want to send multiple requests at the same time, while waiting for all responses concurrently. For these scenarios it is possible to receive responses asynchronously.

By using `BeginRequest`, your thread will not block, and it will permit you to handle the response the way that best suits your application. This method complies with .NET "Asynchronous Programming Model". You can find more information about the "Asynchronous Programming Model" in the Web.

For example, your application can handle responses asynchronously by using a callback, which is a piece of logic that executes asynchronously when the response is received. Define a callback method like this:

```
[C#]

void OnLoginResponseReceived(IAsyncResult result) {
        IMessage response = interactionServerProtocol.EndRequest(result);
```

```
        if (response.Id == EventAck.MessageId)
        {
                var eventAck = (EventAck)response;
                // continue here
        }
        else
        {
                // handle the error here
        }
}
```

Then you can submit your request using the callback method.

```
[C#]

interactionServerProtocol.BeginRequest(loginRequest, OnLoginResponseReceived, null);
```

As an alternative, you may want to do something concurrently, while waiting for the response. For example, you could perform two agent login requests concurrently: one for logging the agent into the T-Server, and another for logging the agent into Interaction Server.

```
[C#]

var resultLoginVoice = tserverProtocol.BeginRequest(loginVoiceRequest, null, null);
var resultLoginMultimedia = interactionServerProtocol.BeginRequest(loginMultimediaRequest,
null, null);

var loginVoiceResponse = tserverProtocol.EndRequest(resultLoginVoice);
var loginMultimediaResponse = interactionServerProtocol.EndRequest(resultLoginMultimedia);

// handle responses, both are available now
```

When using the `BeginRequest` method, your application receives the response to your request as the return value of `EndRequest`. You will not receive the response as an unsolicited event. (You can change this behavior by using the `CopyResponse` protocol property, described below).

CopyResponse

Previously it was stated that responses returned by request methods are not received as unsolicited events by default. This behavior can be modified by using the protocol `CopyResponse` property. The default value is false, but it can be set to true like this:

```
[C#]

tserverProtocol.CopyResponse = true;
```

This is particularly useful for protocols which define events that can be both received unsolicited and as a response to a client request (such as `EventAgentLogin` defined by the T-Server protocol). By setting the `CopyResponse` property to `true`, you can execute your agent state change logic only when handling the message as an unsolicited event, and you do not need to include it when receiving the message as a response.