Platform SDK Developer's Guide

Working with Custom Servers

12/12/2025

Working with Custom Servers

Java

The `ServerChannel` class was designed to give you the ability to develop custom servers using Platform SDK. A `ServerChannel` instance can accept incoming connections, receive and handle incoming messages, and send responses to the clients.

Creating a ServerChannel Instance

Before creating a `ServerChannel` instance, your application should define an instance of some class which implements the `ProtocolFactory` interface. You can use any of the existing Platfrom SDK message factories, although most of these classes cannot follow the logic of existing servers because the messages used in the handshake procedure are hidden.

The most flexible protocol for any extension is `ExternalServiceProtocol`, because it does not require any handshake by default. The following example will use this protocol to create an instance of `ServerChannel`:

```
final ServerChannel server = new ServerChannel(new WildcardEndpoint(11111),
        new ExternalServiceProtocolFactory());
```

Defining Handlers to Process Incoming (Closed) Connections

`ServerChannel` generates two events to manage client connections. When a new client tries to connect, `ServerChannel` raises the `onClientChannelOpened` event, and when a client disconnects `ServerChannel` raises an `onClientChannelClosed` event. Your code can then process these events, as shown in the example below:

```
server.addChannelListener(new ServerChannelListener() {
        public void onClientChannelOpened(OutputChannel channel) { /* … */ }
        public void onClientChannelClosed(ChannelClosedEvent event) { /* … */ }
        public void onChannelOpened(EventObject event) { /* … */ }
        public void onChannelError(ChannelErrorEvent event) { /* … */ }
        public void onChannelClosed(ChannelClosedEvent event) { /* … */ }
});
```

Starting the Server

```
server.open();
```

Processing Incoming Messages

`ServerChannel` supports multiple ways to receive and process incoming messages:

- receiveRequest Method
- External Receiver
- Message Handler

More details about each approach are explored below.

Using the receiveRequest Method

Using this method allows your to define exactly when messages are read. However, you should remember that the internal queue which contains incoming messages is not unlimited. The maximum capacity of this queue will be equal to 4k elements, and once that capacity is filled each new incoming message will cause the oldest one to be lost.

The most popular way of using the `receiveRequest` method is inside a dedicated thread, as shown here:

```
new Thread() {
        @Override
        public void run() {
                while (running) {
                        RequestContext request = server.receiveRequest();
                        if (request != null) {
                                Message requestMessage = request.getRequestMessage();
                                Message respondMessage;
                                // TODO generate respondMessage
                                if (respondMessage != null) {
                                        request.respond(respondMessage);
                                }
                        }
                        Thread.yield();
                }
        }
}.start();
```

Using an External Receiver

To use an external receiver, your should create a class which implements the `RequestReceiverSupport` interface, and then use the `ServerChannel.setReceiver(RequestReceiverSupport receiver)` method to assign this receiver to `ServerChannel`.

The simplest implementation to process incoming messages is shown below:

```
RequestReceiverSupport receiver = new RequestReceiverSupport() {

        public void onChannelOpened(EventObject event) { /* ... */}
        public void onChannelError(ChannelErrorEvent event) { /* ... */}
        public void onChannelClosed(ChannelClosedEvent event) { /* ... */}
```

```
        public void setInputSize(int inputSize) { /* ... */}
        public void releaseReceivers() { /* ... */}
        public int getInputSize() { return 0; }
        public void clearInput() { /* ... */}
        public RequestContext receiveRequest(long timeout) { return null; }
        public RequestContext receiveRequest() { return null; }

        public void processRequest(RequestContext request) {

                Message requestMessage = request.getRequestMessage();
                Message respondMessage;
                 // TODO generate respondMessage
                 if (respondMessage != null) {
                         request.respond(respondMessage);
                 }
        }
};

server.setReceiver(receiver);
```

Using Message Handler

Starting with release 8.5.1, Platform SDK has included a new mechanism to handle incoming messages. `ServerChannel` was extended with a new method `setClientRequestHandler`, that can be used as shown in the following example:

```
server.setClientRequestHandler(new ClientRequestHandler() {

        @Override
        public void processRequest(RequestContext context) {
                Message requestMessage = request.getRequestMessage();
                Message respondMessage;
                 // TODO generate respondMessage
                 if (respondMessage != null) {
                         request.respond(respondMessage);
                 }
        }
});
```

Closing ServerChannel

Closing the server channel causes all active incoming connections to be closed also. To close server channel use the `ServerChannel.close()` method.

```
server.close();
```

.NET

The `ServerChannel` class was designed to give you the ability to develop custom servers using Platform SDK. A `ServerChannel` instance can accept incoming connections, receive and handle incoming messages, and send responses to the clients.

Creating a ServerChannel Instance

Before creating a `ServerChannel` instance, your application should define an instance of some class which implements the `IMessageFactory` interface. You can use any of the existing Platfrom SDK message factories, although most of these classes cannot follow the logic of existing servers because the messages used in the handshake procedure are hidden.

The most flexible protocol for any extension is `ExternalServiceProtocol`, because it does not require any handshake by default. The following example will use this protocol to create an instance of `ServerChannel`:

```
const int portNumber = 22222;
var server = new ServerChannel(new WildcardEndpoint(portNumber),
        new ExternalServiceProtocolFactory());
```

Defining Handlers to Process Incoming (Closed) Connections

`ServerChannel` generates two events to manage client connections. When a new client tries to connect, `ServerChannel` raises the `ClientChannelOpened` event, and when a client disconnects `ServerChannel` raises an `ClientChannelClosed` event. Your code can then process these events, as shown in the example below:

```
server.ClientChannelOpened += (sender, args) =>
{
        var arg = args as NewChannelEventArgs;
        if (arg != null)
        {
                var incomingChannel = arg.Channel;
                // TODO: do something with incoming channel
        }
};
server.ClientChannelClosed += (sender, args) =>
{
        var closedChannel = sender as DuplexChannel;
        var arg = args as ClosedEventArgs;
        var cause = (arg != null)?arg.Cause:null;
        // TODO: process closed channel with known arguments and reason of closing
};
```

Starting the Server

```
server.Open();
```

Processing Incoming Messages

`ServerChannel` supports multiple ways to receive and process incoming messages:

- ReceiveRequest Method

- External Receiver
- Message Handler

More details about each approach are explored below.

Using the ReceiveRequest Method

Using this method allows your to define exactly when messages are read. However, you should remember that the internal queue which contains incoming messages is not unlimited. If messages are kept in the queue for a long time (5 seconds by default, although this value can be changed by setting the PsdkCustomization.ReceiveQueueTimeLimit property) without being read, then the maximum capacity of this queue will be equal to 4k elements and each new incoming message will lead to lose the eldest one.

The most popular way of using the `ReceiveRequest` method is inside a dedicated thread, as shown here:

```
var processMessagesThreadActiveFlag = new ManualResetEvent(false);
var processMessagesThread = new Thread(() =>
{
        while (!processMessagesThreadActiveFlag.WaitOne(100))
        {
                var request = server.ReceiveRequest(TimeSpan.FromMilliseconds(0));
                if (request == null) continue; // nothing to do
                var message = request.RequestMessage;
                IMessage respond = null;
                // TODO: respond = result of process request
                if (respond != null)
                request.Respond(respond);
        }
});
processMessagesThread.Start();

// TODO:

processMessagesThreadActiveFlag.Set();
processMessagesThread.Join();
```

Using an External Receiver

To use an external receiver, your should create a class which implements the `IRequestReceiverSupport` interface, and then use the `ServerChannel.SetReceiver(IRequestReceiverSupport receiver)` method to assign this receiver to `ServerChannel`.

One implementation to process incoming messages is shown below:

```
class ServerRequestReceiver : IRequestReceiverSupport
{
        public void ClearInput(){}
        public int InputSize { get; set; }
        public void ReleaseReceivers(){}
        public IRequestContext ReceiveRequest(){ return null; }
        public IRequestContext ReceiveRequest(TimeSpan timeout){ return null; }
```

```
        public void ProcessRequest(IRequestContext request)
        {
                if (request == null) return; // nothing to do
                var message = request.RequestMessage;
                IMessage respond = null;
                // TODO: respond = result of process request
                if (respond != null)
                        request.Respond(respond);
        }
}
server.SetReceiver(new ServerRequestReceiver());
```

Using Message Handler

Starting with release 8.5.1, Platform SDK has included a new mechanism to handle incoming messages. `ServerChannel` was extended with a new event called `Received`, that can be used as shown in the following example:

```
server.Received += (sender, args) =>
{
        var channel = sender as DuplexChannel;
        var arg = args as MessageEventArgs;
        if (arg == null) return;
        var incomingMessage = arg.Message;
        IMessage outgoingMessage = null;
        // TODO: outgoingMessage = result of processing incomingMessage
        if ((outgoingMessage != null) && (channel != null))
          channel.Send(outgoingMessage);
};
```

Closing ServerChannel

Closing the server channel causes all active incoming connections to be closed also. To close server channel use the `ServerChannel.Close()` method.

```
server.Close();
```