Platform SDK Developer's Guide

Transport Layer Substitution

4/9/2025

Transport Layer Substitution

Java

The Transport Layer Substitution feature, introduced with Platform SDK release 8.5.300.02, allows you to isolate the transport layer and provide alternative ways of transporting messages. It is up to your code to exchange messages and manage connections. Goals of this feature are:

- Simulation of different server behavior in test scenarios.
- Replacement of binary protocol with any other; that is, allowing alternative message delivery sub-systems such as AMQP, MQTT, or STOMP.
- Creation of proxies, hubs etc.

Design Notes

The injected transport layer has to implement the following interface:

ExternalTransport API

```
package com.genesyslab.platform.commons.protocol;

/** Describes API of external transport. */
public interface ExternalTransport {
    /**
     * Connects to a specified destination asynchronously.
     * This method is called during the protocol is opening.
     * Method has to notify {@link ExternalTransportListener#onConnected()}
     * otherwise it must notify {@link ExternalTransportListener#onDisconnected(Throwable)}
     * @param endpoint Endpoint which describes destination address and contains
configuration.
     */
    void connect(Endpoint endpoint);

    /**
     * Disconnects from destination.
     * Method has to notify {@link ExternalTransportListener#onDisconnected(Throwable)}
     */
    void disconnect();

    /**
     * Sends message to the destination.
     * @param message which has to be sent.
     */
    void sendMessage(Message message);

    /**
     * Set external transport listener.
     * @param listener that will handle the transport events.
     */
    void setTransportListener(ExternalTransportListener listener);
}
```

ExternalTransportListener API

```
package com.genesyslab.platform.commons.protocol;

/**
 * API for external transport events handling.
 * <p>
 *     All events have to be notified within some transport notification thread.
 *
 *     Any events can't be notified by transport inside of calls of
 *     {@link ExternalTransport#connect(Endpoint)},
 *     {@link ExternalTransport#disconnect()} or
 *     {@link ExternalTransport#sendMessage(Message)}
 *     or {@link RecursiveCallException} will be thrown.
 * </p>
 */
public interface ExternalTransportListener {

    /**
     * The method is called (using channel's invoker)
     * as soon as the external transport connected to the destination.
     * @throws RecursiveCallException when it is notified inside calls of the methods
     * {@link ExternalTransport#connect(Endpoint)},
     * {@link ExternalTransport#disconnect()},
     * {@link ExternalTransport#sendMessage(Message)}
     * or {@link ExternalTransport#setTransportListener(ExternalTransportListener)}
     */
    void onConnected();

    /**
     * The method is called (using channel's invoker)
     * as soon as the external transport disconnected from the destination.
     * @param cause of disconnection. It is null if the disconnection happened due to call
     * of {@link ExternalTransport#disconnect()}.
     * @throws RecursiveCallException when it is notified inside calls of the methods
     * {@link ExternalTransport#connect(Endpoint)},
     * {@link ExternalTransport#disconnect()},
     * {@link ExternalTransport#sendMessage(Message)}
     * or {@link ExternalTransport#setTransportListener(ExternalTransportListener)}
     */
    void onDisconnected(Throwable cause);

    /**
     * The method is called (using channel's invoker)
     * for each message received by the external transport.
     * @param message that is received by the external transport.
     * @throws RecursiveCallException when it is notified inside calls of the methods
     * {@link ExternalTransport#connect(Endpoint)},
     * {@link ExternalTransport#disconnect()},
     * {@link ExternalTransport#sendMessage(Message)}
     * or {@link ExternalTransport#setTransportListener(ExternalTransportListener)}
     */
    void onMessageReceived(Message message);
}


/**
 * Used to control caller thread and prevent some recursive calls.
 */
public class RecursiveCallException extends RuntimeException {
    public RecursiveCallException(String message) {
        super(message);
    }
}
```

To substitute the transport layer used for a protocol instance in your application, you must complete the following steps:

1. Set either the system property `com.genesyslab.common.protocol.transport.factory` or `PsdkCustomization.PsdkOption.TransportFactoryImpl` to be equal to a fully qualified `ExternalTransportFactory` implementation class name. This helps to control which transport layer will be used for a specified pair (protocol description and endpoint) of arguments.
2. Use the `DuplexChannel.setExternalTransport(ExternalTransport transport)` method. This helps to substitute the transport layer for a specified instance of a channel. An external transport layer (non-null) that is set using `DuplexChannel.setExternalTransport` has more priority then the previous method of substitution.

ExternalTransportFactory API

```
package com.genesyslab.platform.commons.protocol;

/**
 * Factory of external transport for a specified protocol description and an endpoint.
 */
public interface ExternalTransportFactory {

    /**
     * Gets external transport for a specified protocol description and an endpoint.
     * @param protocolDescription specifies a protocol.
     * @param endpoint specifies host, port and configuration.
     * @return external transport instance
     * or null if PSDK have to use the default implementation.
     */
    ExternalTransport getTransport(ProtocolDescription protocolDescription, Endpoint
endpoint);
}
```

Notes:

- The `Connect` operation uses an active Endpoint of the channel, which has its own configuration. Using the settings for encoding, ADDP, TLS and other values is the responsibility of the injected transport layer.
- Sending `Connected`, `Disconnected`, and `ReceivedMessage` events is the responsibility of the injected transport layer. All transport layer implementations notify a listener, but if the notifications are performed directly inside calls to the connect, disconnect, send, or `setTransportListener` methods then `RecursiveCallException` is thrown.

.NET

The Transport Layer Substitution feature, introduced with Platform SDK release 8.5.300.02, allows you to isolate the transport layer and provide alternative ways of transporting messages. It is up to your code to exchange messages and manage connections. Goals of this feature are:

- Simulation of different server behavior in test scenarios.

- Replacement of binary protocol with any other; that is, allowing alternative message delivery subsystems such as AMQP, MQTT, or STOMP.
- Creation of proxies, hubs etc.

Design Notes

The injected transport layer has to implement the following interface:

API of External Transport

```
/// <summary>
/// Describes API of external transport
/// </summary>
public interface IExternalTransport
{
  /// <summary>
  /// Returns state of connection to the destination.
  /// </summary>
  ConnectionState State { get; }
  /// <summary>
  /// Connects to destination.
  /// This method is called during the protocol is opening.
  /// Method has to raise event <see cref="Connected"/> if connection was successful,
  /// otherwise it must raise event <see cref="Disconnected"/>.
  /// <param name="endpoint">Endpoint which describes destination address
  /// and contains configuration.</param>
  /// </summary>
  void Connect(Endpoint endpoint);

  /// <summary>
  /// Disconnects from destination.
  /// Method has to raise event <see cref="Disconnected"/> when connection disconnected.
  /// </summary>
  void Disconnect();

  /// <summary>
  /// Fired, when connection to the destination becomes opened.
  /// </summary>
  event EventHandler Connected;

  /// <summary>
  /// Fired, when connection to the destination becomes closed.
  /// </summary>
  event EventHandler<ConnectionEventArgs> Disconnected;

  /// <summary>
  /// Sends message to the destination
  /// </summary>
  /// <param name="message">Message which has to be sent</param>
  void SendMessage(IMessage message);

  /// <summary>
  /// This event hast to be fired when received IMessage is ready for user.
  /// </summary>
  event EventHandler<MessageEventArgs> ReceivedMessage;
}
```

The injection of a new transport layer is made by using the public method of DuplexChannel class. Its signature is:

```
public void SetExternalTransport(IExternalTransport transport)
```

Notes:

- The `Connect` operation uses an active Endpoint of the channel, which has its configuration. Using settings of encoding, ADDP, TLS etc. becomes a responsibility of the injected transport layer.
- Sending `Connected`, `Disconnected`, and `ReceivedMessage` events is the responsibility of the injected transport layer.

Abstract Implementation

Platform SDK may provide base abstract implementation in order to simplify the 3rd-party code which uses an external transport layer. It may have the following signature:

Abstract Implementation of Basic Functionality

```
/// <summary>
/// Abstract implementation of basic functionality of the
/// <see cref="IExternalTransport"/> interface.
/// </summary>
public abstract class ExternalTransportBase:AbstractLogEnabled, IExternalTransport
{
  /// <summary>
  /// Returns state of connection to the destination.
  /// </summary>
  public ConnectionState State { get; protected set; }

  /// <summary>
  /// Sends message to the destination
  /// </summary>
  /// <param name="message">Message which has to be sent</param>
  public abstract void SendMessage(IMessage message);

  /// <summary>
  /// Connects to the destination. No need to raise any events.
  /// After successful connection property <see cref="State"/> has to be set to <see
cref="ConnectionState.Opened"/> state.
  /// </summary>
  /// <param name="endpoint">Endpoint which describes destination address
  /// and contains configuration.</param>
  /// <exception cref="Exception">If connection is unsuccessful.</exception>
  public abstract void DoConnect(Endpoint endpoint);


  /// <summary>
  /// Disconnects from the destination.
  /// </summary>
  public abstract void DoDisconnect();

  /// <summary>
  /// Fired, when connection to the destination becomes opened.
  /// </summary>
  public event EventHandler Connected;

  /// <summary>
  /// Fired, when connection to the destination becomes closed.
  /// </summary>
  public event EventHandler<ConnectionEventArgs> Disconnected;

  /// <summary>
  /// This event hast to be fired when received IMessage is ready for user.
  /// </summary>
  public event EventHandler<MessageEventArgs> ReceivedMessage;

  /// <summary>
  /// Simulates receiving message.
  /// </summary>
  /// <param name="message">Message which is received.</param>
  public void OnMessageReceived(IMessage message) {…}

  void IExternalTransport.Disconnect() {…}

  void IExternalTransport.Connect(Endpoint endpoint) {…}

  /// <summary>
  /// Raise <see cref="Disconnected"/> event.
  /// </summary>
  /// <param name="args"><see cref="ConnectionEventArgs"/> parameter.</param>
  protected void FireDisconnect(ConnectionEventArgs args) {…}

  /// <summary>
  /// Raise <see cref="Disconnected"/> event.
  /// </summary>
  /// <param name="args"><see cref="ConnectionEventArgs"/> parameter.</param>
  protected void FireConnect(EventArgs args) {…}
}
```

Usage of this implementation allows end-user simplify own implementation.

Code Sample

Example of External Transport Implementation

```
internal class TheNewTransport : IExternalTransport
{
  private readonly IMessageFactory _factory = new ConfServerProtocolFactory();
  public ConnectionState State { get; private set; }
  public void Connect(Endpoint endpoint)
  {
    State = ConnectionState.Opening;
    Console.WriteLine("Connecting to: {0}", endpoint.ToString());
    ThreadPool.QueueUserWorkItem(state =>
    {
      Thread.Sleep(100); //
      // TODO: do something to connect to destination
      var handler = Connected;
      State = ConnectionState.Opened;
      if (handler != null) handler(this, null);
    });
  }
  public void Disconnect()
  {
    State = ConnectionState.Closing;
    Console.WriteLine("Dicsonnecting");
    ThreadPool.QueueUserWorkItem(state =>
    {
      Thread.Sleep(100);
      // TODO: do something to disconnect from destination
      var handler = Disconnected;
      State = ConnectionState.Closed;
      if (handler != null) handler(this, null);
    });
  }
  public event EventHandler Connected;
  public event EventHandler<ConnectionEventArgs> Disconnected;
  public void SendMessage(IMessage message)
  {
    Console.WriteLine("Sending message: {0}", message);
    if (message.Id == 49) // request ‘RequestProtocolVersion’
    {
      var msg = _factory.CreateMessage(50);
      // response ‘EventProtocolVersion’
      msg["ReferenceId"] = message["ReferenceId"];
      msg["OldProtocolVersion"] = message["ProtocolVersion"];
      ReceiveMessage(msg);
    }
    if (message.Id == 3) // request ‘RequestRegisterClient’
    {
      var msg = _factory.CreateMessage(19);
      // response ‘EventClientRegistered’
      msg["ReferenceId"] = message["ReferenceId"];
      msg["OldProtocolVersion"] = message["ProtocolVersion"];
      msg["ProtocolVersion"] = message["ProtocolVersion"];
      ReceiveMessage(msg);
    }
  }
  private void ReceiveMessage(IMessage message)
  {
    var handler = ReceivedMessage;
    if (handler != null)
    {
      var args = new MessageEventArgs(message);
      handler(this, args);
    }
  }
  public event EventHandler<MessageEventArgs> ReceivedMessage;
}
```

Using Injected Transport (Test Method Snippet)

```
var client = new ConfServerProtocol(new Endpoint("localhost", 56789))
client.Timeout = TimeSpan.FromSeconds(3);
client.SetExternalTransport(new TheNewTransport());
client.Open();
Assert.IsTrue(client.State == ChannelState.Opened);
client.Close();
Assert.IsTrue(client.State == ChannelState.Closed);
```

Example Using Base Implementation

```
internal class CustomExternalTransport : ExternalTransportBase
{
  public override void SendMessage(IMessage message)
  {
    Console.WriteLine("Message received: {0}",message); // log to console
    IMessage returnMessage = null;
    // TODO: process handshake and other logic
    if (returnMessage!=null)
      OnMessageReceived(returnMessage); // returns message to sender
  }
  public override void DoConnect(Endpoint endpoint)
  {
    // TODO: connect (or do nothing if there is no need to connect anywhere)
    State = ConnectionState.Opened;
  }
  public override void DoDisconnect()
  {
    // TODO: disconnect
  }
}

[TestMethod]
public void TestExternalTransport()
{
  var protocol = new ConfServerProtocol(new Endpoint("localhost",12345));
  protocol.SetExternalTransport(new CustomExternalTransport());
  protocol.Open();
  Assert.IsTrue(protocol.State == ChannelState.Opened);
  var request = RequestReadLocale.Create(123);
  var response = protocol.Request(request);
  Assert.AreSame(request,response);
  protocol.Close();
  Assert.IsTrue(protocol.State==ChannelState.Closed);
}
```

Dynamically Linked Factory

An external transport layer may be linked to an existing application dynamically by using the factory interface implementation contained in the external assembly. This interface has the following description:

IExternalTransportFactory description

```
/// <summary>
/// Describes API of external transport factory.
/// It may be dynamically linked and used by ClientChannel before opening.
/// </summary>
public interface IExternalTransportFactory
{
  /// <summary>
  /// Creates instance of external transport for given endpoint.
  /// </summary>
  /// <param name="protocolDescription">Description of protocol</param>
  /// <param name="endpoint">Endpoint which is used as key to create external
transport</param>
  /// <returns>Instance of external transport, or null if it is not needed</returns>
  IExternalTransport GetTransport(ProtocolDescription protocolDescription, Endpoint endpoint);
}
```

The public class which implements this interface must have a public constructor with no parameters. Otherwise it won't load correctly.

This factory provides allows you to create different implementations - not only for different protocols, but also even for different instances of the same protocol - by using the `protocolDescription` and `endpoint` parameters.

The location of your assembly and class name should be defined in the app.config file before your application starts:

App.Config Configuration File

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add key="ExternalTransport.AssemblyFileName" value="[path]\..."/>
    <add key="ExternalTransport.ClassName" value="..."/>
  </appSettings>
</configuration>
```