Axini Modeling Platform

ADAPTER TRAINING

– LABORATORY –

Introduction

In this training, we will develop an *adapter* to connect the Axini Modeling Platform (AMP) to a system under test (SUT). The particular SUT is a standalone implementation of the familiar SMARTDOOR application, which was tested in a previous Axini modeling course. In fact – when the adapter is finished – you will be able to test the standalone SMARTDOOR SUT with the AML models that you previously constructed.

The 'Axini Plugin Adapter' protocol neither does prescribe a programming language nor a platform on which the adapter will be executed. The protocol, however, relies on two technologies: the WEBSOCKET protocol for communication and the Google Protocol Buffers (PROTOBUF) framework for serializing and deserializing messages. For most mainstream programming languages there are open source libraries available for both technologies.

Consequently, the training does not prescribe a programming language. In the training, however, we will use a Java implementation of an adapter as a reference implementation. Hence, any programming language (except Java) can be selected to be used in this course.

This laboratory text describes a *possible* sequence of exercises to arrive at a complete plugin adapter. You are free to implement certain parts of the adapter in a different order, though; it is sometimes more natural to simultaneously work on several classes at the same time. Furthermore, the development of the different classes of a plugin adapter can be split into independent tasks, which can be done by different people.

Background. We expect that the participants of the training are fluent in the programming language of their choice. This means that concepts like object-oriented programming, generics, socket communication, multi-threaded programming, etc. should be familiar ground.

Naming conventions. In this training we will use the `camelCase` convention for the names of identifiers; this is due to our reference implementation in Java. For the implementation of the adapter which you are about to build, it is best to comply to the conventions of your implementation language.

Version: 0.2 (December 19, 2022)

0. Preparations

0.1 Axini files

Apart from the WEBSOCKET and PROTOBUF libraries (see below), there are some some additional files needed for this training:

- standalone SMARTDOOR SUT: supplied as a Java program,
- Axini's PROTOBUF `.proto` files, defining the messages to be exchanged between AMP and the adapter, and
- the complete Java implementation of a plugin adapter for the standalone SMARTDOOR SUT: to be used for reference and inspiration.

In preparation for the training you should do the following:

- Check that you have all resources: this file and the three files listed above.
- Ensure that you can execute the SMARTDOOR SUT.

0.2 PROTOBUF

Google Protocol Buffers (PROTOBUF) is an open-source cross-platform data format used to serialize structured data. It is typically used to communicate over a network or for storing data.

Google PROTOBUF's website is https://developers.google.com/protocol-buffers. Here you can read about PROTOBUF, download the `protoc` compiler or study the PROTOBUF tutorial for your programming language.

In preparation for the training you should do the following:

- Download and install the PROTOBUF library for your programming language of choice.
- Use the `protoc` compiler to generate the source files for the `.proto` message files of the 'Axini Plugin Adapter' protocol (see above).
- Browse the tutorial for your chosen language.

0.3 WEBSOCKET

WEBSOCKET is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. See https://en.wikipedia.org/wiki/WebSocket.

For most programming languages, there exist (open-source) libraries to ease the communication using the WEBSOCKET protocol. See appendix A for a list of well-known WEBSOCKET libraries. If your company has already standardized on a WEBSOCKET library, it is best to use that library.

In preparation for the adapter training you should do the following:

- Select the WEBSOCKET library that you are going to use for your adapter. The libraries mentioned first in the lists of appendix A are strong candidates.
- Download and install the selected library.

- Run some included tests to ensure that the library is properly installed.
- Browse the documentation to get a feeling of the WEBSOCKET library.

0.4 Logging framework

An adapter is typically a multi-threaded program, which can make it hard to analyze and debug the flow of the complete application. A good logging framework can be very helpful here. Your software company may have already standardized on a logging framework. Feel free to install your favorite logging framework for this training.

1. First steps

Before we are going to build the actual plugin adapter, we start with some exercises to get comfortable with both PROTOBUF and WEBSOCKET.

1.1 Create PROTOBUF messages

The generated PROTOBUF files provide several methods to construct PROTOBUF objects for the adapter protocol. Construct a program, which:

- constructs a `Label` for the stimulus `'open'`,
- constructs an empty `Configuration`,
- constructs a `Configuration` object with a single configuration item,
- constructs a `Message` object containing a `Label`, and
- prints all the constructed objects on the `stdout`.

Refer to the online PROTOBUF documentation to see how messages can be constructed on the basis of `.proto` messages.

1.2 WEBSOCKET connection to SmartDoor SUT

In this exercise, we are going to connect to the standalone SMARTDOOR SUT, which behaves like a WEBSOCKET server. Over this connection only text messages (Strings) are being exchanged. Construct a program, which:

- connects to the SMARTDOOR SUT,
- sends a `RESET` message to the SUT, and
- observes the `RESET_PERFORMED` message from the SUT.

2. WEBSOCKET connection to AMP

Now that we can construct PROTOBUF messages and know how to connect to a WEBSOCKET server, we can take the plunge and make a connection with AMP. Again, we do not yet create separate classes, but construct a small program which performs some steps.

2.1 Connect to AMP

AMP's `adapter-broker` acts as a WEBSOCKET server. The difference with the standalone SMARTDOOR is that the messages are now *binary*: serialized PROTOBUF messages.

Furthermore, the server of the `adapter-broker` requires an authorization *token* during the inital handshake of the connection. This token can be found on the adapter page of AMP.

When connecting to the AMP, the adapter has to provide a (key, value) pair in the connection request, where the key is `'Authorization'` and the value is `'Bearer '` plus the token of AMP.

2.2 Send announcement to AMP

Construct a PROTOBUF message for an `Announcement`. An `Announcement` consists of three parts (see also announcement.proto):

- `name`: name of the adapter,
- `configuration`: settings for this adapter, and
- `labels`: all supported stimuli and responses of the SUT.

The `name` has to be unique for AMP and typically consists of the name of the adapter and the name of the machine on which the adapter is run (e.g., `smartdoor@foobar`).

The `configuration` will hold the configuration settings for this adapter (e.g., the URI of the SUT), which can be altered on the adapter page of AMP. For now, we can leave the configuration empty.

AMP need to know all labels which are supported by the adapter, to check whether the adapter is compatible with the AML model. This means that we have to construct a list of PROTOBUF Labels. For now, we also leave this part of the `Announcement` empty.[1]

Use the encoding functionality of PROTOBUF messages to serialize the `Announcement` as a array of bytes and send it to AMP.

2.3 Receive configuration from AMP

After receiving the announcement from the adapter, AMP will register the adapter and shows its availability on AMP's adapter page. On this adapter page, the user of AMP can edit the configuration of the adapter.

When a test is started in AMP, AMP will send the (updated) configuration to the adapter. This configuration message is also a byte array and should be decoded by the PROTOBUF library. The adapter can use the received configuration to configure the SUT.

In this exercise – because we have sent an empty configuration within the announcement – the configuration from AMP will also be empty.

[1] The list of supported labels can be left empty in an announcement. AMP can still communicate with the adapter but cannot check the compatibility beforehand.

3. Complete adapter

Next we are going to generalize the fragments of the previous exercises into classes to build a complete adapter. We use the Java implementation of the adapter for guidance and inspiration.

The adapter will consists of several parts, each implemented by a single class:
- `BrokerConnection`: takes care of the WEBSOCKET connection with AMP.
- `AdapterCore`: keeps the state of the adapter, handles the PROTOBUF messages, and calls the appropriate methods of the `BrokerConnection` and `Handler`.
- `Handler`: abstract base class which defines the methods which *should* be implemented by the SUT specific handler.
- `SmartDoorHandler`: starts and stops the connection with the SUT, forwards messages from AMP to the SUT, and vice-versa.
- `SmartDoorConnection`: takes care of the WEBSOCKET connection with the standalone SMART-DOOR SUT.
- `AxiniProtobuf`: class which contains helper methods to construct PROTOBUF messages; is independent of any SUT.

The classes `BrokerConnection`, `AdapterCore`, `Handler` and `AxiniProtobuf` are independent of the SUT. These classes can be reused for all future SUTs.

3.1 Handler

Before completing the AMP side of the adapter, we should define the abstract class `Handler` which defines the methods that the SUT specific part of adapter *must* implement. The `Handler` class provides methods that the `AdaperCore` can call.

See the class `Handler` of the Java adapter for the methods which need to be provided by each handler. The most important (abstract) methods are:
- `start`: connect to the SUT and ensure that the SUT is in it is initial state, ready to run,
- `stop`: stop the current test,
- `reset`: prepare for a next test case, and
- `defaultConfiguration`: returns the default configuration for this SUT.

3.2 BrokerConnection

The `BrokerConnection` class is responsible for the WEBSOCKET connection with the `adapter-broker` of AMP. Remember that AMP's `adapter-broker` acts as the WEBSOCKET *server* and accepts connections requests from clients.

Preferably, the WEBSOCKET library takes care of all low-level details of the connection. Then, we only have to specify what should happen when the connection is *opened* or *closed*, and what should happen when a text or binary *message* is received from AMP.

The WEBSOCKET library usually takes care of spawning a separate thread which consumes WEBSOCKET messages and then calls `onMessage`.

3.3 AdapterCore

The class `AdapterCore` holds the state of the adapter: disconnected, connected, announced, configured or ready. It provides a method `handleMessage` which parses a PROTOBUF message and then calls the appropriate method.

Construct the `AdapterCore` class and make sure that when a message is received from AMP, the appropriate methods are called. For now, you can focus on the 'happy flow' behavior of AMP and do not yet consider all error scenarios.

3.4 AxiniProtobuf

The generated PROTOBUF files have all the functionality to construct and manipulate PROTOBUF objects. We experienced – for our Ruby and Java adapters – that the construction of PROTOBUF message can be tedious and cumbersome.

Create a class `AxiniProtobuf` which will contain some wrapper methods to easy the construction of PROTOBUF objects. Create methods to construct the following PROTOBUF objects on the basis of data values of your programming language:

- `Label`
- `Parameter`
- `Configuration`
- `Message`

Refer to `AxiniProtobuf.java` for the list of wrapper methods which might be needed.

3.5 SmartDoorHandler

Now that most of the SUT independent parts of the plugin adapter are in place, it becomes time to fill in the SUT specific parts. For this we need to create the class `SmartDoorHandler`, a subclass of the abstract class `Handler`.

At least, all abstract methods of `Handler` need to be implemented:

- `start`: initiate a WEBSOCKET connection to the SUT, and send a 'RESET' to reset the SUT,
- `stop`: close the WEBSOCKET connection to the SUT, and
- `reset`: send a 'RESET' to the SUT.

An important part of any SUT specific handler is the conversion of messages: either from PROTOBUF messages (stimuli) to SUT messages or from observed SUT messages (responses) to PROTOBUF messages. For the SMARTDOOR SUT the conversion is easy. The standalone SUT expects UPPERCASE commands and replies with UPPERCASE reponses. So, the label `open` has to be converted to `OPEN` and the SUT response `INVALID_COMMAND` has to be converted to a PROTOBUF label whose name is `invalid_command`.[2]

Finally, the class `SmartDoorHandler` knows about all the labels that the SUT supports. The method `getSupportedLabels` constructs this list of labels as a list of PROTOBUF `Label` objects. This method uses the wrapper methods of `AxiniProtobuf`.

[2] In practice, the conversion is usually not so easy. For instance, when the SUT messages are XML messages. Manually constructing such a complete list would be tedious and errorprone. In such cases, the converters are typically generated from the message definition specifications (such as XSDs) or from AML label definitions.

3.6 SmartDoorConnection

Finally, we have to implement the WEBSOCKET connection to the standalone SMARTDOOR SUT. This is similar to the `BrokerConnection`: the SUT acts as a WEBSOCKET server. The communication itself is easier though: only text messages (Strings) are being exchanged.

3.7 Ultimate Test

You should now have a complete adapter to connect AMP to the SMARTDOOR SUT. Use your AML models of the original SMARTDOOR exercise to test your adapter and the SMARTDOOR SUT.

4. Bonus

Although your adapter may be working correctly for the *happy flow* of both, it is not yet feature complete. We could add several aspects to the adapter to make the adapter more robust or improve the testing power of the AMP/adapter combination.

- Add a default configuration to `SmartDoorHandler` with the address of the SUT and the manufacter SUT to be used.
- Add a virtual stimulus `reset`, which can be used to reset the SUT from within the AML model.
- Add support for virtual stimuli to inject bad weather messages into the SUT. For example: non-UTF8 messages, binary messages, very large messages, etc.
- Add error handling to `AdapterCore` to react on bad weather messages from AMP.
- Your adapter will be probably terminate after a single test run or when an error has occured. Make sure that your adapter always stays alive.

A. Some WEBSOCKET libraries

Below we have listed some well-known WEBSOCKET libraries for different programming languages. You are free to choose your own favorite library, though.

Java:
- Nathan Rajlich's Java-WebSocket https://github.com/TooTallNate/Java-WebSocket
  Barebones WEBSOCKET server and client implementations, 100% Java. Used for the Java reference implementation of the plugin adapter.
- (Eclipse) Tyrus https://eclipse-ee4j.github.io/tyrus/
  Java API for easy development of WEBSOCKET applications.

C++:
- WebSocket++ https://www.zaphoyd.com/projects/websocketpp/
  Popular, header only C++ library.
- Boost.Beast https://github.com/boostorg/beast
  Another popular C++ library. Requires Boost(.Asio) and C++ 11.
- Libwebsockets (LWS) https://libwebsockets.org
  Flexible, lightweight pure C (not C++) library.

C#:
- websocket-sharp https://github.com/sta/websocket-sharp
  Convenient wrapper offering event handling methods.
  Available from within Visual Studio as a NuGet package.
- Native C#: `System.Net.WebSockets`
  Low level WEBSOCKET implementation which requires more work (multi-threading, synchronisation, etc.).

Python:
- websocket-client https://github.com/websocket-client/
  WEBSOCKET client (only) implementation offering event handling methods; has extensive documentation.
- websockets https://github.com/aaugustin/websockets
  Built on top of `asyncio`, offers an elegant coroutine-based API, but no callback event handling.

Ruby:
- faye-websocket https://github.com/faye/faye-websocket-ruby
  Used within Axini for in-house adapters.
- EM-WebSocket https://github.com/igrigorik/em-websocket
  EventMachine based, WEBSOCKET server; used within the Ruby SMARTDOOR SUT.