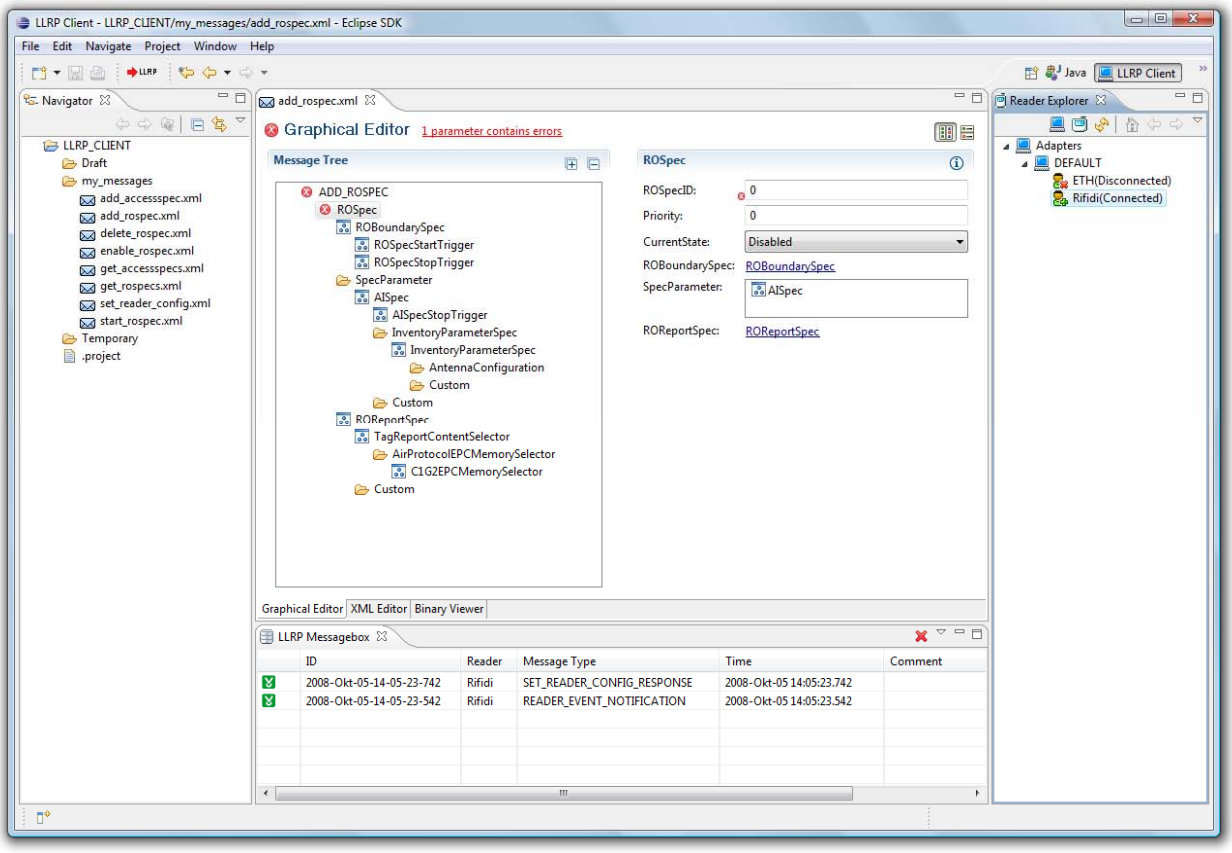
Fosstrak LLRP GUI Client



Ulrich Etter, Samuel Wieland, Haoning Zhang

Supervisors: Christian Floerkemeier, Christof Roduner

Distributed Systems Lab Project, ETH Zurich, October 8th 2008

Table of Contents

Introduction

Objective

The objective of the **LLRP GUI Client** Project is to implement a simple GUI Client that allows users to assemble, send and receive LLRP messages without programming. It allows the maintenance of several LLRP readers in parallel running either locally or distributed on remote instances.

As a long-term goal the LLRP GUI Client is intended to replace all proprietary GUI tools for LLRP readers.

Background and Motivation

LLRP stands for low level reader protocol and specifies a protocol for the control of RFID readers (see http://www.epcglobalinc.org/standards/llrp for the standards and for a more detailed introduction). LLRP is called low level because it enables the client to finetune many parameters on the RFID reader like the radio power, the antennas present and even the modes of operation of the RFID reader. LLRP is not to be confused with the reader protocol (RP). RP is also a communication protocol for RFID readers but opposed to LLRP it operates on a fairly high level of abstraction.

The communication between two LLRP endpoints runs through a binary protocol which is efficient and fast. The LLRP GUI project uses LTKJava to compile LLRP messages from binary to java objects, from java objects to xml and from xml to binary.

The motivation for the LLRP GUI client arises from basically 4 facts/problems:

1. Tedious/complicated messages that are hard to assemble.

   There are many dependencies within LLRP messages a user has to take care of (some parameters require to lie within a specific interval, others require the presence/absence of parameters, ...). With a missing parameter the resulting binary message is invalid and cannot be read/parsed by the RFID reader. The reader will answer with an error message in binary format. For the user this implies tedious try-and-error until the message has the correct format. The LLRP GUI client gives here the user the ability to assemble different LLRP messages with either a graphical editor or textually with an xml editor. All messages are checked dynamically at runtime for missing parameters or for invalid entries. Thatwise the user receives immediate feedback and can correct errors before sending the message.

2. Binary protocol is hard to debug

   When an LLRP message does not perform the task intended (for example when the user wants to enable an RO_SPEC and by accident choses the wrong RO_SPEC id) it is very hard to find out the reason for the misbehaviour. The only way how a user can debug is to inspect and validate all the message parameters by hand. This means to split a possibly very long bitstring of ones and zeros into the numerous message parts. The LLRP GUI client has the ability to transform the binary string automatically into a more human readable form (like an xml message or like a tree displaying all the parameters as leaves). It is even possible to correct erroneous parameters on the fly.

3. Many different RFID readers with proprietary interfaces but with a standardized LLRP protocol.

   As there are many different RFID readers from different vendors available it is not feasible to configure these readers through the client applications delivered by the vendor. Consider the

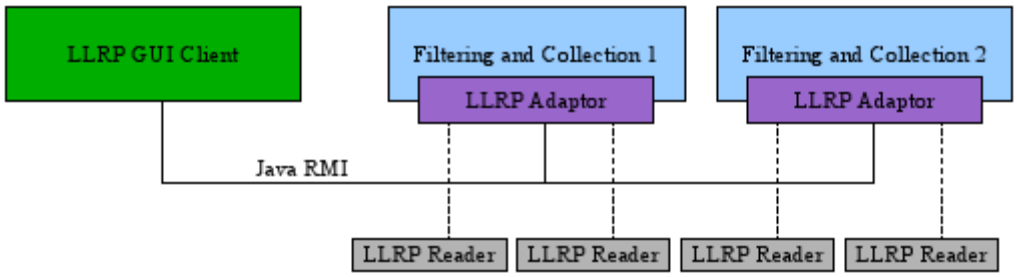
example where a client has 10 readers from 10 different vendors with a completely incompatible configuration application for each of the readers. With LLRP as a standardized protocol for the communication the LLRP GUI client gives the possibility to configure and maintain all of the 10 readers with one single tool simplifying life a great deal.

4. Many messages

   LLRP is a low level protocol. It is therefore not surprising that there are many messages exchanged between the reader and the client. The user needs some help in receiving and storing those messages. The LLRP GUI client receives the messages for the user and stores them separately for each of the registered readers. The messages are clearly flaged by their message type such that the user can easily distinguish between usfull messages and messages that are not relevant.

Integration into Fosstrak

The LLRP GUI client is a part of the Fosstrak project. It has been developed in such a way that it can interact with other existing Fosstrak applications. It has been a goal of the project to enrich the existing application layer events middleware (filtering and collection) with the ability to communicate with LLRP enabled RFID readers. Most parts of the communication layer of the LLRP GUI client can be used without modification directly by Fosstrak filtering and collection.



For a more detailed introduction please refer to the respective chapters in the developers guide:

Filtering and collection:

- Logical Reader Concept
- How to implement an Adaptor

LLRP GUI client:

- AdaptorManagement

Developer Guide

This chapter addresses the needs of developers that want to maintain or extend the LLRP GUI Client. If you want to get an overview of what the system does, you might want to have a look at the User Guide in the appendix first.

The developers guide is divided into several subchapters.

- GUI: All the graphical user interface parts.
- Management: Essentials of the networking part.

GUI

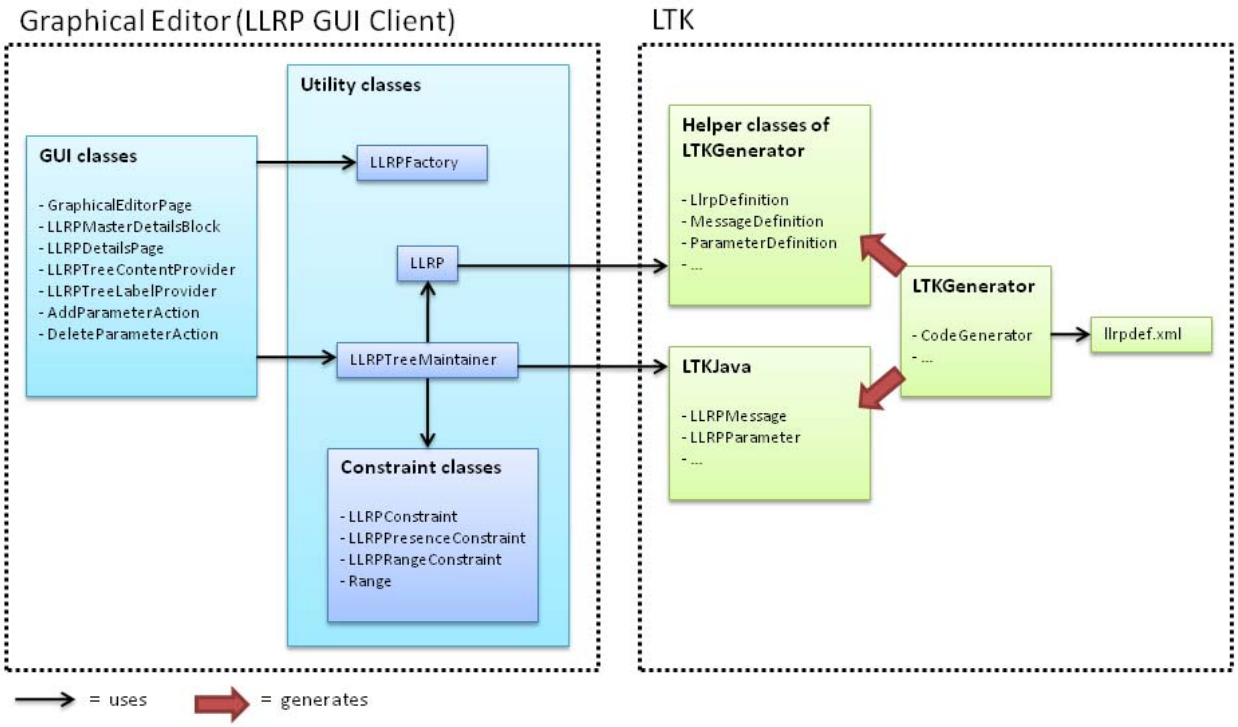
The GUI chapter shall give you an overview to the implementation of all the GUI elements.

It is structured into 6 subchapters:

- Graphical Editor
- XML/Binary Editor
- Message Box
- Navigator
- Reader Explorer
- Wizard

Graphical Editor

This section speaks about the design of the graphical editor. The following diagram gives you an overview of its architecture:



GUI Classes

The graphical editor is realized using Eclipse Forms. It implements the "master/details" user interface design pattern. The "master" part consists of a tree that shows the structure of the message. The "details" part displays the fields and sub-parameters of the parameter that is currently selected in the master part.

For detailed information have a look at the following classes:

- org.accada.llrp.client.editors.graphical.GraphicalEditorPage
- org.accada.llrp.client.editors.graphical.LLRPMasterDetailsBlock
- org.accada.llrp.client.editors.graphical.LLRPDetailsPage
- org.accada.llrp.client.editors.graphical.LLRPTreeContentProvider
- org.accada.llrp.client.editors.graphical.LLRPTreeLabelProvider
- org.accada.llrp.client.editors.graphical.actions.AddParameterAction
- org.accada.llrp.client.editors.graphical.actions.DeleteParameterAction

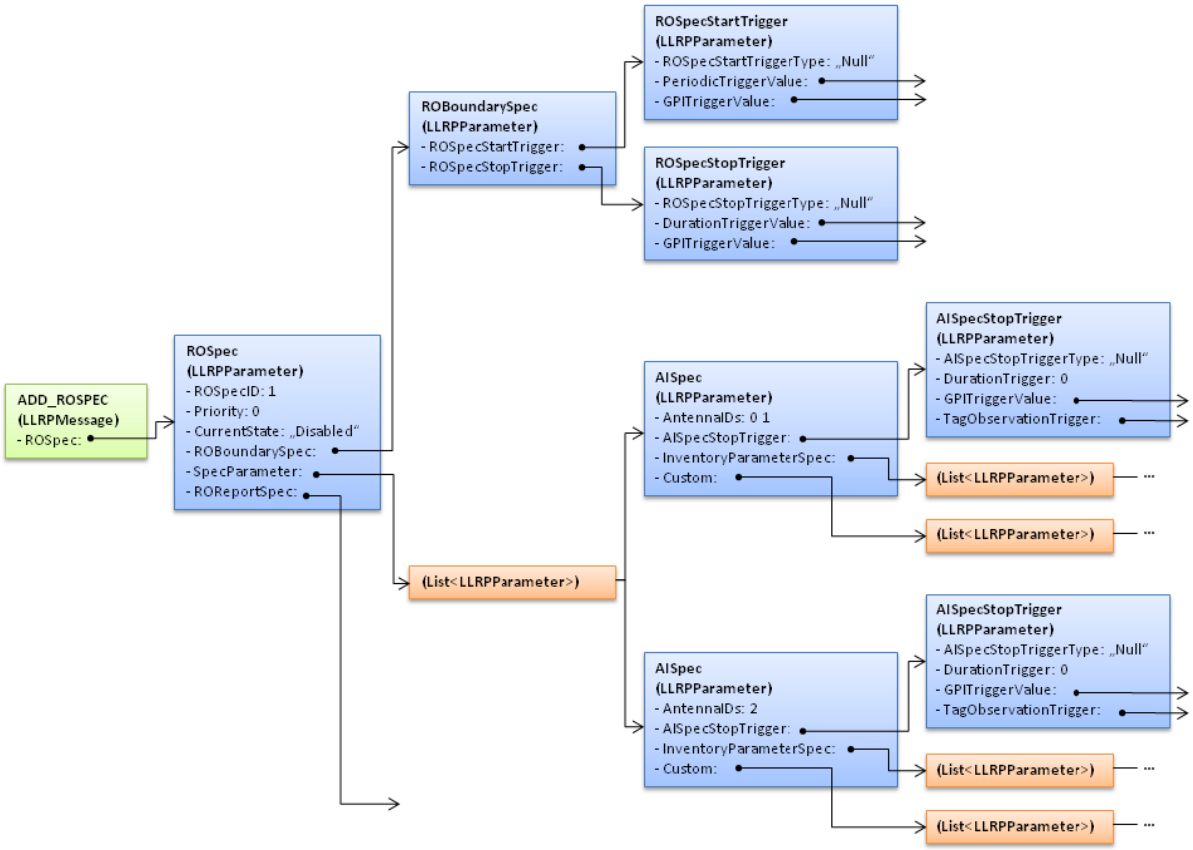
Utility Classes

The classes described in this section are all used by the graphical editor, but in principle they are independent of the GUI classes and can be used irrespective of them (The class `LLRPFactory`, for example, is also used by the New Message Wizard). We are talking about:

- org.accada.llrp.client.util.LLRPTreeMaintainer
- org.accada.llrp.client.util.LLRP
- org.accada.llrp.client.util.LLRPFactory
- org.accada.llrp.client.util.LLRPConstraints
- org.accada.llrp.client.util.LLRPPresenceConstraint
- org.accada.llrp.client.util.LLRPRangeConstraint
- org.accada.llrp.client.util.Range

Underlying model

As model classes for the graphical editor the classes of LTKJava are used (`LLRPMessage`, `LLRPParameter`, etc.). This is, the graphical editor works on a tree of LTKJava objects. The root of the tree is an object of type `LLRPMessage`, and has `LLRPParameter`s and lists of `LLRPParameter`s as children. As an example, consider the following picture that shows the (slightly simplified) object graph for a typical ADD_ROSPEC message:



We call the nodes of this graph **tree elements**. A tree element is either of type `LLRPMessage`, `LLRPParameter` or `List<LLRPParameter>`. Moreover, every tree element has 0 or more **fields**, 0 or more **children** (which can be null), and exactly 1 **parent** (the parent of the root is null). The tree element `ROSpec` (which is of type `LLRPParameter`), for instance, has 3 fields (`ROSpecID`, `Priority`, `CurrentState`), 3 children (`ROBoundarySpec`, `SpecParameter`, `ROReportSpec` (null)) and the tree element ADD_ROSPEC as parent.

Because LTKJava does not provide a method to get all children of a tree element, the children are retrieved indirectly by using information from `llrpdef.xml` [1], and **reflection**. For example, to get the children of the `ROSpec` parameter, the names of the children (`ROBoundarySpec`, `SpecParameter`, `ROReportSpec`) are learned by querying `llrpdef.xml`, and then the corresponding getter-methods of `ROSpec` (`getROBoundarySpec()`, `getSpecParameterList()`, `getROReportSpec()`) are called using reflection.

All functionality to navigate and modify the LTKJava object graph is encapsulated in the class `LLRPTreeMaintainer`.

The code to query the `llrpdef.xml` is contained in the class LLRP (to be precise: the class LLRP does not directly access `llrpdef.xml`, but uses helper classes of the LTKGenerator [2]).

[1] `llrpdef.xml` is the LLRP description in XML, see http://llrp-toolkit.wiki.sourceforge.net/LLRP+Protocol+Definition+(llrpdef.xml).

[2] The LTKGenerator is the module that generates the LTKJava classes from llrpdef.xml.

Validation functionality

When a user edits a message in the graphical editor, he gets immediate feedback about the correctness of the message in form of error flags and error messages. This functionality is enabled by methods in the class `LLRPTreeMaintainer` that allow to validate fields and parameters (e.g. `validateField(...)` and `validateChildPresence(...)`).

Additional LLRP constraints not modeled in llrpdef.xml

The LLRP specification defines some constraints on fields and parameters which are not (yet) modeled in `llrpdef.xml`. For instance, it specifies that the field `Priority` of the `ROSpec` parameter must have a value between 0 and 7 (**Range Constraint**). Another example is that the sub-parameter `PeriodicTriggerValue` of `ROSpecStartTrigger` must be present when the field `ROSpecStartTriggerType` is set to `Periodic` (**Presence Constraint**). Because these constraints will probably get integrated into `llrpdef.xml` in the long term, it was not worth the effort to come up with our own data format to define these rules. To still provide the user as much help as possible to edit messages in the graphical editor, we currently just hard-coded these constraints in the Java code. For examples of how to specify such constraints have a look at the classes `LLRPConstraint`, `LLRPRangeConstraint` and `LLRPPresenceConstraint`.

Information on messages and parameters

In the graphical editor, the user can get a description of the currently selected parameter by clicking on the information icon. This functionality is realized by reading these descriptions from `llrpdef.xml` (see method `getDescription(...)` in class `LLRP`) and using the SWT browser widget to show the HTML content nicely formated to the user (see method `createDescriptionAction()` in class `LLRPDetailsPage`).

XML Editor/Binary Viewer

Editor

The XML Editor extends the Eclipse editing environment, the functions include:

- enables the user to edit the LLRP XML files loaded from the Repository
- dynamically transfers the XML-format LLRP message to Java Objects through LTK-Java API and passes to Graphical Editor and Binary Viewer

  The whole Editor framework is generated by Eclipse Plug-In Assistant. Here is the short description of generated files:

  Class **ColorManager.java** maintains the color table used in XML-based editors to hihight different tags.

  Class **ConfigrationModel.java** keeps the information of whole list of reserved words (tags). The model are passed to LLRPContentAssistant.java.

  Class **LLRPContentAssistant.java** provides Auto-Completion functions.

  Class **IXMLColotConstants.java** includes constant color values used for sections in LLRP XML files.

  Class **MessageElement.java** represents tree node in the XML file.

  Class **TagRule.java** recognizes the Tags part in the XML file.

  Class **XMLPartitionScanner.java** separates Partitions in the XML file.

  Class **XMLTagScanner.java** recognizes the Token part in the XML file.

  Class **XMLWhitespaceDetector.java** recognizes the White space in the XML file.

  Class **XMLScanner.java** implements one XML Parser.

Binary

The Binary Viewer is a read-only view embeded in the Eclipse's editing area. It only receives the updated LLRP message and reflects its binary format, but user cannot modify its value.

When user switchs from **XML Editor** or **Graphical Editor** to **Binary Viewer**, the LLRPEditor module firstly generates the LLRPMessage instance, if there is any critical (Exception caught), switching to the **Binary Viewer** is disabled. This validation work guarantees that the user always sees a valid binary interpretation of the message.

In memory the binary message stores as a BinaryMessage instance, which is transformed by **BinaryMessageHelper.java**. The BinaryMessage holds attributes of binary strings (**BinarySingleValue** class), which assembles the whole binary message.

Message Box

Message Repository provides a stable storage interface for incoming LLRP messages. It stores not only the XML-Format LLRP messages, but also manages the key attributes of the message (like message type, receiving time, from which reader, etc) as indexes.

Interface

The interface **Repository** defines all operations to maintain the Message Logs. After **ResourceCenter** module initializes the instance of the **Repository**, the Repository registers itself in the Reader Management Module. When new messages come from the reader, the Reader Management Module calls back the put() function to store the message in XML text format.

Note: If the incoming message is not valid, the Reader Management Module will catch the exception first and will not pass it to the **Repository**. In other word, the stored messages are always valid.

The functions in the interface include:

- `open()`
- `put()`
- `get()`
- `getTopN()`
- `clearAll()`
- `close()`
- `public void open()`

  How the resource layer (File System, Database, and etc.) initializes its resources. This function will be called when the client starts.

  `public void close()`

  How the resource layer (File System, Database, and etc.) closes and destroys related resource handlers. This function will be called when the client exits.

  `public LLRPMessageItem get(String aMsgSysId)`

  Get the LLRP Message Item from repository according to the unique Message ID.

  `public void put(LLRPMessageItem aMessage)`

  Put the LLRP Message Item to the repository.

  `public ArrayList<LLRPMessageItem> getTopN(int aRowNum)`

  Get the Top N LLRP Message items (desc by issue time). Please notice that the returned objects only includes metadatas for the MessageList view, there is NO message content in those objects because of performance concern.

  `public void clearAll()`

  Clear all the items in repository.

JavaDB

The LLRP message repository implementation based on Sun JavaDB. Please make sure the derby.jar in the build path before you can start the database.

We implemented the JavaDB Repostory on Sun JavaDB (oringinally Derby) engine. When JavaDBRepostory starts up, it checks the existence of defined database **llrpMsgDB**. If it doesn't exist, it creates the database instance and the table **LLRP_MSG**.

view

The MessageBox view follows the Eclipse Label/Content Provider pattern. **MessageItem.java** represents The single entry in the table view; it records the meta-data of the message (without XML Content). **MessageboxViewLabelProvider.java** extends the LabelProvider, provides column text/image interpretation. **MessageBoxView.java** act as one ViewPart, it holds one Eclipse TableViewer.

Navigator

Project Navigator leverages the existing Eclipse Project Navigator view, to manage the LLRP XML messages and its templates in Eclipse workspaces.

When one message is retrieved from the Message Repository, which is triggered by users, the message would be written to a new file (Refer the writeMessageToFile() in ResourceCenter.java). The new generated file will be showed and managed in the Project Navigator view.

HealthCheck

The Project Navigator enables users to specify the Eclipse Project which holds the LLRP message file.

By default, especially in the case when user installed the Plug-In and starts in first time, the Eclipse project **LLRP_CLIENT** is required. To simplify the configuration task for the users, the Plug-In provides one Health Check feature to detect the Eclipse configuration. This function makes the client starting properly, even without configuration.

When Eclipse loads the LLRP Client, the Health Check function is triggered to validate whether the environment is ready for use. If any significant error is detected (for example, the folders corrupted or database table doesn't exist), the report form will pop up at startup.

```
HealthCheckDialog dlg = new HealthCheckDialog(
        PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell());

if (!dlg.getHealthCheck().validate()) {
        dlg.open();
}
```

HeathCheck class is designed in a **Chain of Responsibility** pattern. It maintains the list HealthItem class. When the Plug-In starts, it registers the check items in the HealthCheck. Each HealthItem inplements two functions: validate() and fix().

```
public boolean validate();

public void fix();
```

At first, the HealthCheck iterates all the items and run their validate() one by one. Those validate() compiles the errors when they occur the problems. Then, as the user requests, the HealthCheck can iterate all marked items, which includes health problems, and execute those fix() function to solve the problem.

In current release, there are two CheckItem provided.

- CheckEclipseProject.java, which checks the existence of Eclipse Project and its subfolders, and
- CheckRepository.java, which validates the JavaDB as message holder.

Reader Explorer

Reader Explorer manages the LLRP adapters & reader resources in the development environment. It enables users to add/remove/import the reader information.

On the GUI side, it presents as an Eclipse tree view. The external resources are hierarchically organized as tree. All adapters are the first level children, and each adapter owns several physical readers.

View

Class **ReaderExplorerView** includes one Eclipse **TreeViewer**, and illustrates the Adapter/Reader relationship in tree-like hierarchy. The node object presented in the tree is implemented as **ReaderTreeObject** class, which defines its connection attributes and its child nodes as well.

The **ReaderExplorerView** follows the Eclipse's Content/Label Provider pattern. Class **ReaderExplorerViewContentProvider** maintains the information of readers in memory. In additional, it synchronizes the reader profile with AdaptorManagement module. Class **ReaderExplorerViewLabelProvider** provides String or Image handle when the node labels show themselves. The labels, string and image, are context-sensitive, that means the labels and images will be different against device type (adapter or reader) and connection status (connected or disconnected).

ContextMenu

One context-sensitive pop-up Menu is provided in ReaderExplorerView. According the status of selected reader, the menu includes 5 Eclipse Actions to implement **connect**, **disconnect**, **remove**, **GetReaderConfig** and **GetROSpec** commands.

Wizard

The "New LLRP Message" wizard is a standard JFace wizard. It consists of two classes, `org.accada.llrp.client.wizards.NewLLRPMessageWizard` and `org.accada.llrp.client.wizards.NewLLRPMessageWizardPage`, that extend the corresponding eclipse classes. `NewLLRPMessageWizard` is the main class and specifies what happens when a user presses the "Finish" button. It has a `NewLLRPMessageWizardPage` that defines the input fields and handles user input.

To generate a new LLRP message the wizard uses the `LLRPFactory`.

Management

The Management chapter shall give you an overview to the implementation of all the background reader and adaptor management facilities.

It is structured into 3 subchapters:

- AdaptorManagement
- Adaptors
- Readers

The AdaptorManagement has been developped in a modular way allowing it to be used without the GUI client. One of the standalone usages is within the LLRP driver for the fosstrak application layer events middleware (filtering and collection).

Adaptor Management

This chapter shall give an overview to the adaptor management providing access to adaptors and readers. The behaviour of the adaptor management is explained by explaining the most important methods and their behaviour.

This includes:

- error management
- message management
- concurrency
- adaptor management
- configuration management

Initialization

```
public boolean initialize(
                String readConfig,
                String storeConfig,
                boolean commitChanges,
                LLRPExceptionHandler exceptionHandler,
                Repository repository)
        throws LLRPRuntimeException { ... }
```

It is **very important** to initialize the adaptor management properly. For this task the initialize method is provided.

The first two parameters tell the adaptor management where to read or write the configuration.

If you want the adaptor management to run a snapshot configuration set the flag `commitChanges` to true. If so, all the changes to the adaptors and to the readers in the local adaptors will be reflected **immediately** back into the configuration file.

The exception handler and the repository are responsible for error and message reporting. If you set them to **null** you will have to use the corresponding setters to specify them in a later phase.

Shutdown

```
public synchronized void shutdown() { ... }
```

If you shutdown your client, it is best to run this method to shutdown the adaptor management. This method performs several cleanup routines, stores the configuration and stops the worker threads.

Definition

```
public synchronized String define(String adaptorName, String address)
  throws LLRPRuntimeException, RemoteException, NotBoundException { ... }
```

In a first step the adaptor management inspects the address parameter. If this parameter is set to **null** the adaptor is assumed to be a local adaptor. Otherwise the address is considered to be a valid ip address of a remote rmi machine.

In the case where the address is not **null** the adaptor management tries to establish an rmi connection to the specified ip address and tries to aquire the adaptor from the rmi registry. Please notice that in the remote case the adaptorName you provided will be overridden by the name of the remote adaptor.

Then the management makes sure that in the local repository no other adaptor with the same name exists.

In order to receive asynchronous messages from the adaptor and therefore also from the readers a asynchronous callback receiver has to be installed. The adaptor management uses the class AdaptorCallback for this purpose.

If all the previous steps have been performed successfully a new worker thread is created and started. The thread takes over the control of the adaptor.

Undefinition

```
public synchronized void undefine(String adaptorName)
    throws LLRPRuntimeException { ... }
```

The adaptor management stops the worker thread, deregisters the asynchronous callback and removes the adaptor from the adaptor list.

Enqueuing

```
public void enqueueLLRPMessage(String adaptorName,
                      String readerName,
                      LLRPMessage message)
                      throws LLRPRuntimeException { ... }
```

In order to simplify the sending of an LLRPMessage the adaptor management provides a short-cut. When you post an LLRPMessage together with the readerName and the adaptorName the adaptor management will select the correct adaptor and send the message asynchronously to the specified reader. One of the main advantages of this procedure is that you do not have to wait for the message to be sent.

MessagePosting

```
public void postLLRPMessageToRepo(LLRPMessageItem message) { ... }
```

If an LLRPMessage arrived on any of the adaptors, this method can be invoked. The adaptor management then takes care of the delivery of the message to the repository.

This also ensures, that if the repository has not been set correctly in advance, the error message will be posted to the error logger.

ExceptionPosting

```
public void postException(
                LLRPRuntimeException e,
                LLRPExceptionHandlerTypeMap
                exceptionType,
                String adapterName,
                String readerName)
{ ... }
```

Whenever an error occurs the clients (adaptors, readers...) can invoke this method. If an exception handler has been installed during setup of the adaptor management, an exception is constructed and is delivered to this exception handler.

In case that the exception handler is not set, an error is reported to the error logger.

LoadFromFile

```
public synchronized void loadFromFile()
    throws LLRPRuntimeException { ... }
```

The adaptor management gives the possiblity to retrieve the adaptors/readers from file. For the storage of the configuration the java.util.Properties are used.

StoreToFile

```
public synchronized void storeToFile() throws LLRPRuntimeException {...}
```

As with loadFromFile this method stores a configuration to a configuration file on disc.

ExceptionHandler

```
public void setExceptionHandler(LLRPExceptionHandler exceptionHandler) {
... }
```

In order to receive asynchronous exceptions from the adaptors and the readers you will have to set an exception handler. If you did not set the handler during initialization you can use this helper method to perform this task.

Repository

```
public void setRepository(Repository repository) {
```

In order to receive LLRP messages in the LLRP GUI Client the respective message repository has to be specified. If you did not set the repository during initialization you can use this helper method to perform this task.

Sample

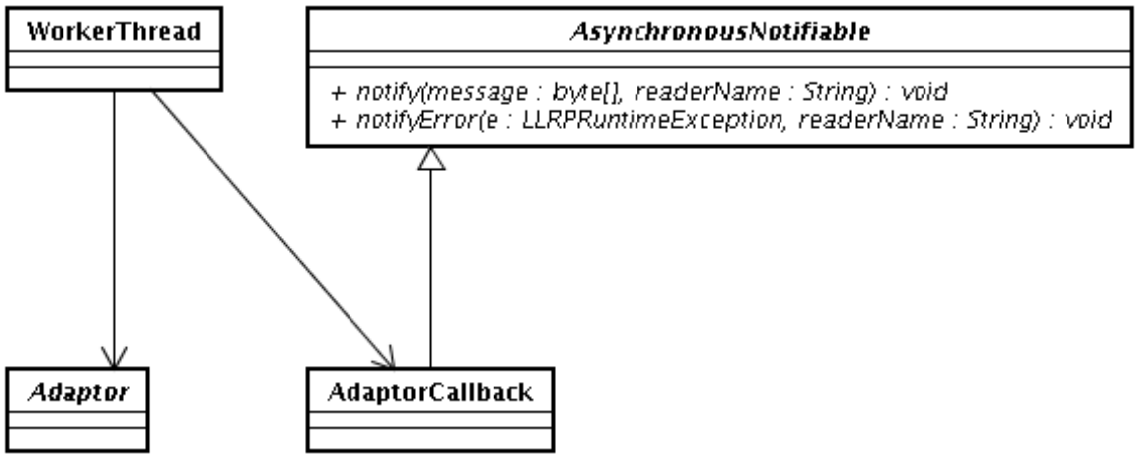
```
// create a message repository
Repository repository = new MessageRepository();
```

```
// create an exception handler
ExceptionHandler handler = new ExceptionHandler();

// run the initializer method
String readConfig =
Utility.findWithFullPath("/readerDefaultConfig.properties");
String writeConfig = readConfig;
boolean commitChanges = true;
AdaptorManagement.getInstance().initialize(
    readConfig, storeConfig, commitChanges, handler, repo);

// now the management should be initialized and ready to be used

// create an adaptor
String adaptorName = "myAdaptor";
AdaptorManagement.getInstance().define(adaptorName, "localhost");

// create a reader
String readerName = "myReader";
Adaptor adaptor = AdaptorManagement.getAdaptor(adaptorName);
adaptor.define(readerName, "192.168.1.23", 5084, true, true);

//Enqueue some LLRPMessage on the adaptor
AdaptorManagement.enqueueLLRPMessage(adaptorName, readerName, message);

// when you shutdown your application call the shutdown method
AdaptorManagement.getInstance().shutdown();
```

Multithreading

The LLRP GUI Client allows several adaptors to be run simultaneously, therefor the AdaptorManagement needs some mechanisms to support a parallel execution of different adaptors.

Each adaptor runs on a thread `AdaptorWorker`. Each thread maintains a callback for the asynchronous message callback.



Adaptors

Adaptors provide an interface for the creation, modification and deletion of LLRP readers.

For each physical LLRP reader the adaptor creates a logical reader that will then maintain the connection. For more details refer to the chapter Readers.

If you want to know more about the creation of an adaptor instance please refer the chapter `Adaptor Management`.

One Implementation of the interface is the class AdaptorImpl. Within this guide we will refer to this reference implementation.

This guide will explain the basic behaviour by explaining the most important methods available in the adaptor interface.

LocalAdaptor

The adaptor runs locally on the same machine as the LLRP GUI Client. In this modes all the calls to the adaptor are performed locally.

RemoteAdaptor

This mode allows the adaptor to be run on a different machine than the LLRP GUI Client (eg. in a filtering and collection environment). That for the adaptor gets exported through java rmi. In this mode of operation a stub gets aquired from the java rmi registry and all calls are performed remotely through this stub.

The only thing that a user has to take care of is the ip address of the adaptor (only for the creation of the adaptor). The remote adaptor should behave in exactly the same way as the local adaptor.

Definition

```
public void define(String readerName,
                String readerAddress,
                int port,
                boolean clientInitiatedConnection,
                boolean connectImmediately)
                throws RemoteException, LLRPRuntimeException;
```

With this method you can create a new reader on this adaptor. In the first step a check is performed whether the specified reader name already exists on the current adaptor.If not a new instance of a reader is generated.

There exist two different connection models specified by LLRP.

- client initiated: in this model the client tries to establish the connection to the reader.
- reader initiated: in this model the client waits for a reader to establish the connection.

You can select the profile according the boolean `clientInitiatedConnection`.

The boolean `connectImmediately` gives you the possibility to delay the connection establishment between the reader-stub and the physical reader. If you set it to false you will have to establish the connection through the reader interface at a later time before using the reader.

Attention: LLRP connections are a 1-to-1 relation. This means one physical reader can only open exactly **one** connection to a client.

Undefinition

```
public void undefine(String readerName) throws RemoteException,
LLRPRuntimeException;
```

When you call this method the reader gets disconnected first and then removed from the adaptor.

Sending

```
public void sendLLRPMessage(String readerName, byte[] message)
    throws RemoteException, LLRPRuntimeException;
```

To send an LLRP message to one of the readers you can use this short-cut message. Just provide the reader name and the message.

Remark: We advise you to use the adaptor management to enqueue messages instead of this sending facility. The reason is that a sending through this method is blocking (synchronous) whereas the sending through the adaptor management is (non-blocking) asynchronous.

Callback

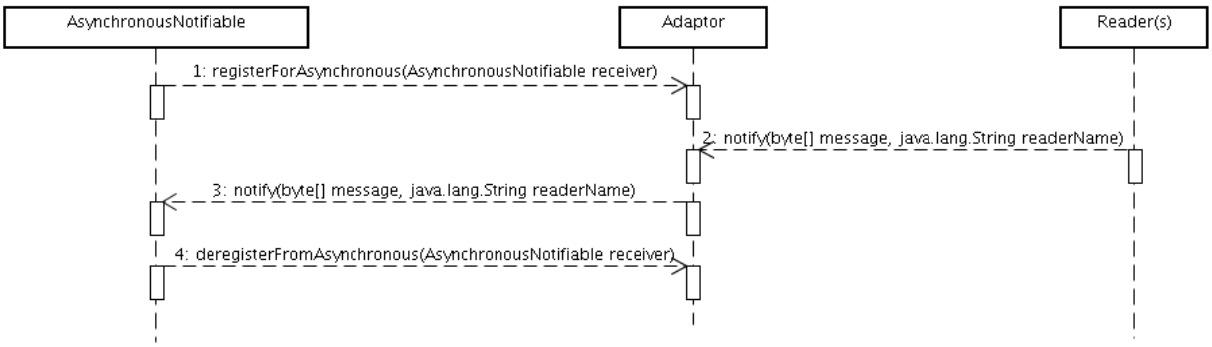
Adaptors support a asynchronous callback mechanism for exceptions and for incoming LLRP messages. Before you can use the asynchronous reporting facility you need to register the interface `AsynchronousNotifiable` on the adaptor. Whenever an exception occurs or when a LLRP message arrives the notifier will be updated and be provided with the message/exception.

Notice: When you create an adaptor through the adaptor management this callback mechanism is already setup for you and you will not need to perform any additional steps.

The registering and deregistering can be done with the following methods:

```
// registering
public void registerForAsynchronous(AsynchronousNotifiable receiver)
    throws RemoteException;

// deregistering
public void deregisterFromAsynchronous(AsynchronousNotifiable receiver)
    throws RemoteException;
```

The following graphic shall give you an overview to the mechanism of the asynchronous callback mechanism.



ErrorReporting

Whenever an error occurs on one of the readers or on the adaptor the error callback is invoked.

```
public void errorCallback(LLRPRuntimeException e, String readerName)
    throws RemoteException;
```

Basically this corresponds to a observer-Pattern. The observable (in this case the adaptor) notifies all the observers (in this case all the registered asynchronous notifiables). The observable triggers the method `notifyError`.

MessageReporting

Whenever an LLRP message arrived from one of the readers, the message callback is invoked.

```
public void messageReceivedCallback(byte[] message, String readerName)
                        throws RemoteException;
```

As with the error callback this is comparable to the observer- pattern. The observable triggers the method notify on all the observers.

Readers

The Reader-interface provides access to the physical LLRP reader. It maintains the connection, handles errors, delivers and retrieves LLRP messages.

In the following guide the implementation of the reader interface ReaderImpl is discussed and used as a reference.

ConnectionEstablishment

```
public void connect(boolean clientInitiatedConnection)
                        throws LLRPRuntimeException, RemoteException;
```

There are two different types of connections that can be used for the communication between the physical reader and the logical reader:

- **Logical reader initiated connection**: The logical reader initiates the connection and tries to connect to the physical reader.
- **Physical reader initiated connection**: The logical reader creates a connection acceptor that waits for incoming LLRP connections. To establish an LLRP connection the physical reader has to initiate the connection.

The user can specify the desired connection profile during the connection setup by specifying the boolean `clientInitiatedConnection`. If set to true the first connection model is selected (the logical reader connects to the physical reader). If set to false the second model is chosen.

Note: If you create the reader through the adaptor interface (**recommended**) and if you specified `connectionImmediately` you will not have to call the connect method as the adaptor will take care of this for you.

As soon as the connection to the physical reader is established a connection watchdog is installed. This watchdog instructs the LLRP reader to periodically send a *KEEP_ALIVE* message. If this message is not received for a certain time interval, the connection is assumed to be **dead**. The reader then removes all connection profiles and triggers an exception.

ConnectionTeardown

```
public void disconnect() throws RemoteException;
```

The client closes the LLRP connection.

Sending

```
public void send(byte[] message) throws RemoteException;
```

The reader checks the message for validity (correct encoding, valid LLRP message...). Then the message is sent to the physical reader. When an unknown exception is triggered the reader usually disconnects.

Remark: We advise you to use the adaptor management to enqueue messages instead of this sending facility. The reason is that a sending through this method is blocking (synchronous) whereas the sending through the adaptor management is (non-blocking) asynchronous.

Callback

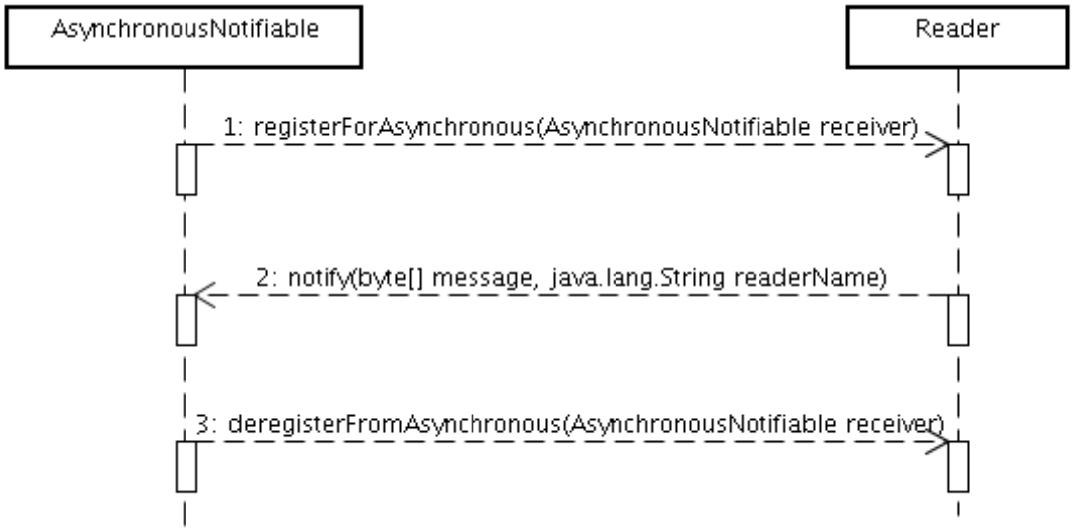
Readers support a asynchronous callback mechanism for exceptions and for incoming LLRP messages. Before you can use the asynchronous reporting facility you need to register the interface AsynchronousNotifiable on the reader. Whenever an exception occurs or when a LLRP message arrives the notifier will be updated and be provided with the message/exception.

Notice: When you create an reader through the adaptor management this callback mechanism is already setup for you and you will not need to perform any additional steps.

The registering and deregistering can be done with the following methods:

```
// registering
public void registerForAsynchronous(AsynchronousNotifiable receiver)
    throws RemoteException;

// deregistering
public void deregisterFromAsynchronous(AsynchronousNotifiable receiver)
    throws RemoteException;
```

The following graphic shall give you an overview to the mechanism of the asynchronous callback mechanism.



ErrorReporting

Whenever an error occurs on the reader the error callback is invoked. As the reader uses ltk to send/receive messages the error callback is slightly different to the one in the adaptors.

```
public void errorOccured(String message) {
  throws RemoteException;
```

Basically this corresponds to a observer-Pattern. The observable (in this case the reader) notifies all the observers (in this case all the registered asynchronous notifiables). The observable triggers the method notifyError.

MessageReporting

Whenever an LLRP message arrived from on the reader, the message callback is invoked. For the same reason as with the error callback also the message reporting callback is slightly different to the one in the adaptor.

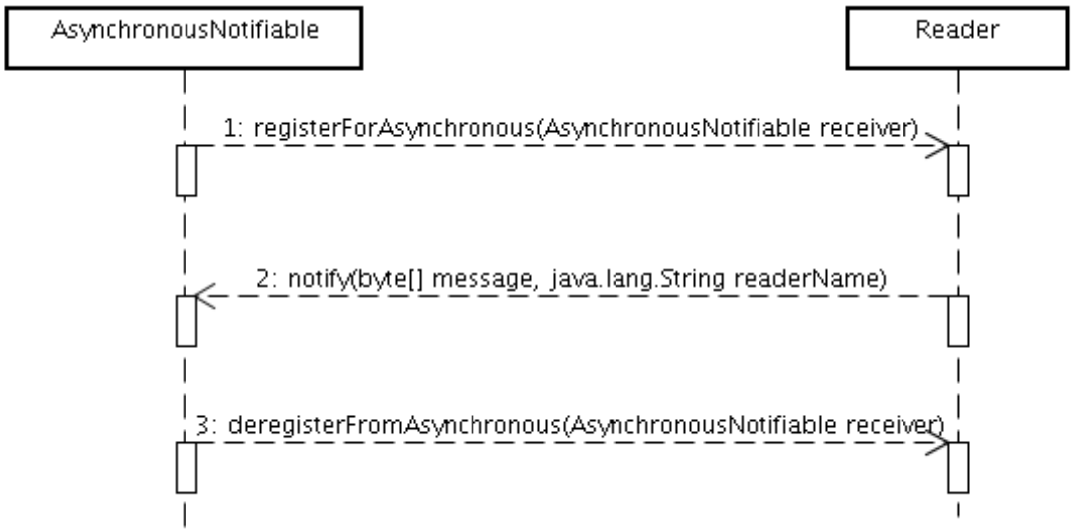
```
public void messageReceived(LLRPMessage message);
```

As with the error callback this is comparable to the observer- pattern. The observable triggers the method notify on all the observers.

The standard configuration of the reader is **not** to report the *KEEP_ALIVE* messages (they occur very often!). However you can instruct the reader to report also those messages.

Asynchronous Notification

The Reader-class exports a simple interface to register for asynchronous message notification. If one is interested into all incoming LLRP messages, one can register a callback through the method void `registerForAsynchronous(AsynchronousNotifiable receiver)`. As soon as an LLRP message arrives on the reader the method void `notify(byte[] message, java.lang.String readerName)` is invoked. In case of an error the error message is reported through the method void `notifyError(LLRPRuntimeException e, java.lang.String readerName)`.



Appendix – User Guide

On the following pages you will find the user guide for the LLRP GUI Client.