



Technologies for Information Systems, A.Y. 2008–2009

Low level architecture of the PERLA middleware

Project by:

MAESANI Andrea	Matr. 719697
MAGNI Claudio Patrizio	Matr. 720827
PADULA Emanuele	Matr. 719348

Contents

1	Introduction	1
1.1	PERLA	1
1.1.1	The language	2
1.1.2	The middleware	2
1.2	Naming conventions and definitions	3
2	Project goals	6
2.1	Requirements	8
3	Proposed architecture	9
3.1	Differences with initial documentation	11
3.2	Guidelines for future developments	11
4	Software design	12
4.1	Class diagrams	12
4.1.1	Package device.adapter	12
4.1.2	Package device.channel	16
4.1.3	Package device.channel.concrete.socket	17
4.1.4	Package device.channel.frame	18
4.1.5	Packages device.channel.frame.address and device.channel.frame.address.concrete	19
4.1.6	Package device.channel.frame.header and device.channel.frame.header.concrete.simple	20
4.1.7	Package device.channel.frame.receiver	21
4.1.8	Package device.channel.frame.sender	22
4.1.9	Package device.fpc	23
4.1.10	Package device.fpcfatory	24

4.1.11	Package configurator	25
4.1.12	Package utils.messages	26
4.2	Guidelines for future developments	28
5	Testing	29
5.1	The Socket Channel	30
5.2	A complete network	31
5.3	Two Gateways in waterfall fashion	39
6	Conclusions	46

Chapter 1

Introduction

This document will explain our work for the project of *Technologies for Information Systems* course.

This chapter will give the reader a general overview of the PERLA System [1] [3], as it was conceived and developed at the beginning of this project. It will lay the foundations to fully understand the following chapters, as our work deals with the core of PERLA System.

1.1 PERLA

PERLA (*PERvasive LAnguage*) is a declarative query language for pervasive systems. It has been designed to support heterogeneity and to be easily deployable on most networks of data collecting devices (i.e. wireless sensor networks). The SQL-like syntax and semantics result in a flat learning curve for users already experienced with other query languages.

The PERLA project aims at developing a complete system, composed by a complex middleware [5] to fully support the language. The system architecture foresees many software layers to let the user see the network of devices almost like a traditional database. The basic idea is to facilitate the compatibility towards new technologies, by providing a middleware structure that spans all the way to the low level, right above the devices.

An application layer (upper interface) is responsible for the interaction with the user and works via queries. A middleware handles all the internal logic of the system, providing to the language an abstraction of the devices

through logical objects. Finally a lower interface is dedicated to the connection of heterogeneous devices to the middleware.

1.1.1 The language

The language supports three kind of queries [4].

Low Level Queries are executed on a single logical object and define how and when the sampling should be performed, how sampled data should be locally processed, and which results have to be produced. A particular feature introduced at this level is the *PILOT JOIN* operation [4], whose aim is to allow dynamic changes in the set of logical objects executing a specific query, based on the results produced by another running query.

High Level Queries are very similar to the normal streaming databases ones and they allow to manage different streams produced by low level queries.

Finally, *Actuation Queries* are not intended to collect data from a logical object, but rather to send commands to the logical object they are executed on.

See [2] for more details.

1.1.2 The middleware

The middleware (Figure 1.1) deserves a little more attention, since it is the object of this project. The goal of the middleware is to provide an abstraction for each device in terms of logical objects and to support the execution of PERLA queries. It's important to specify that we will use the term "logical object" to refer to the complete architecture that allows the system to represent and manage a device.

The *language parser* receives textual queries as input, verifies their syntax and transforms them in a suitable format for distribution and execution.

The high-level implementations of each logical object are called *Functionality Proxy Component* (FPC). High level queries are executed by the high level query executor which is a streaming database engine. The process is more complex for low level queries: firstly the FPC registry is used to find the set of devices that will be involved in the query execution. Then, each involved low level query executor starts sampling its device (through the ab-

straction provided by the FPC) and managing sampled data as required by the query.

The FPC related software is implemented using JAVA technology and each FPC is assumed to be reachable via TCP/IP. The communication protocol between the FPC and the physical device is managed by a specific software layer provided with the middleware.

Each FPC is tightly connected to a Low Level Query Executor (LLQE), which makes use of the FPC to retrieve the needed data and compute the query results.

Another important component is a C library that has been developed to minimize the low level programming effort by the user to integrate a new technology in the middleware. To reach this goal the structure of a XML file is defined, containing a full description of a device in terms of available sensors, measures that can be sampled and packets format. The effort required to the user to integrate a new technology is limited to the creation of the XML descriptor and the extension of the C library with the definition of device specific sampling routines. The FPC wrapping the device is then automatically and dynamically generated by the middleware.

1.2 Naming conventions and definitions

We list here the technical terms related to this particular project, in order to disambiguate possible misunderstandings.

Component: a software object which encloses a specific function in the system

Device: a physical device connected to the system that is able to gather data from the environment

Sensor: a synonym for device (we'll use device as it sounds more general)

Channel: a component whose goal is to abstract a physical communication channel

Middleware: layer of software containing several components that provides the abstraction of the devices to the upper layer software

FPC: (Functionality Proxy Component) a component that represents a device and provides its abstraction

Adapter: a component that acts as a bridge between the channels and the FPC objects

Virtual Channel: a communication system created between a device and its paired FPC

VCI: Virtual Channel Identifier

BVCI: Binding Virtual Channel Identifier

Registry: the component responsible for the registration and the research of FPCs

Binding: the phase in which the device asks to be registered into the system, after it has been connected

Network: the network composed by devices and nodes dedicated to the routing (so the network “under” the middleware)

Gateway: a component that handles the routing of messages among channels inside a network

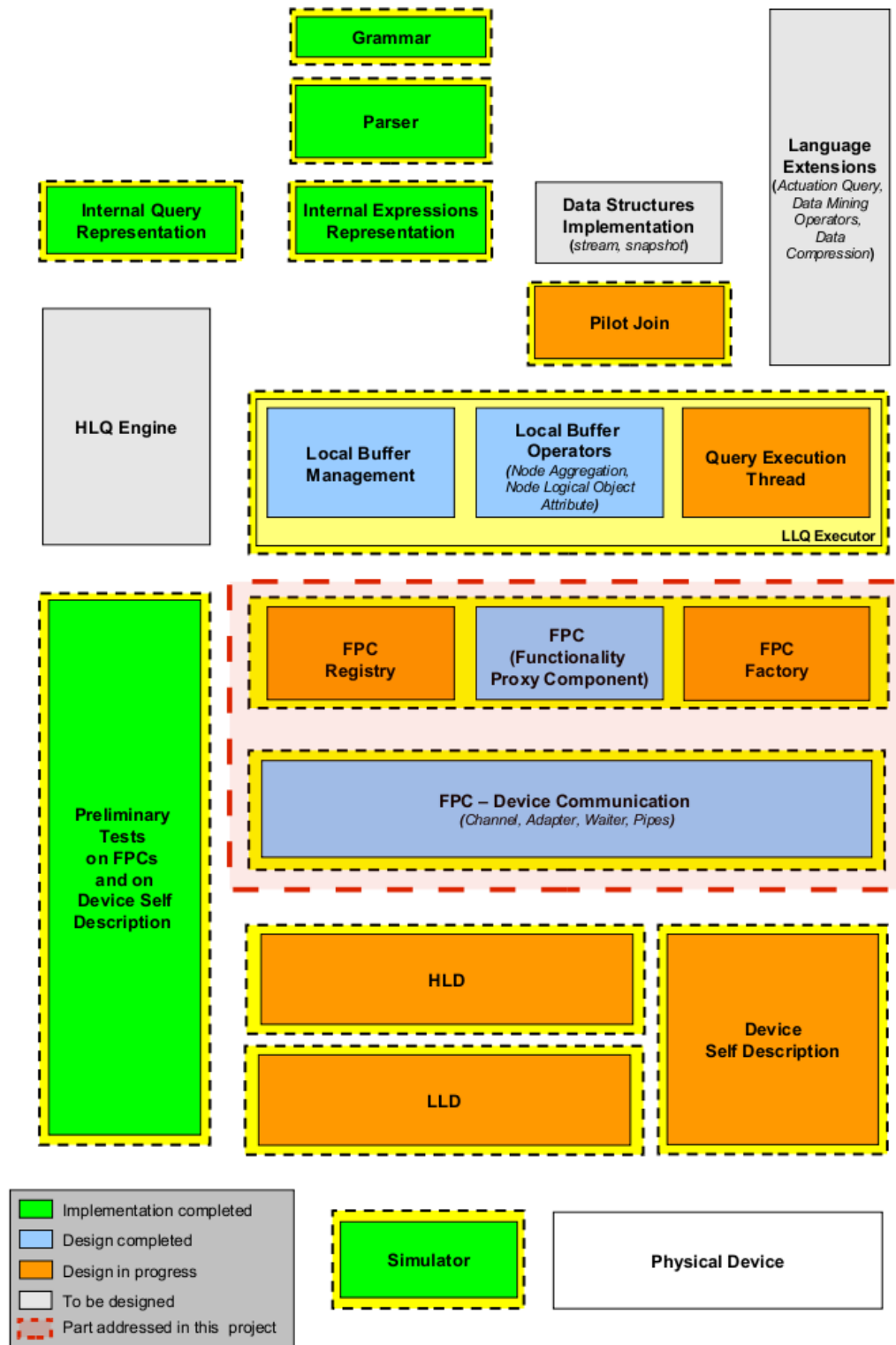


Figure 1.1: Middleware architecture.

Chapter 2

Project goals

The goals of the project were the design and development of the lower part of the PERLA middleware, that is the software layer responsible for the abstraction of the devices and their communication. In particular we focused on the channels, the messages and the adapters. From the implementation point of view, the aim was to make the C part (on the devices) be able to communicate with the Java layer on top of it. We dealt mainly with the syntactic aspect of the communication, addressing only slightly the semantic one (the content of the communication). To better understand where this part fits in, we'll give a visual representation.

The blue box in Figure 2.1 contains the components we decided to develop. The upper layer at the top of the figure is out of our scope and is responsible for the queries execution.

The first need was the design of a way to handle the network of devices. This could be done with the creation of channels among nodes, messages to communicate, gateways to route the messages and a channel manager able to handle/create/destroy channels based on the system configuration.

The second need was the design of adapters able to use the channels to connect the devices. These components should handle virtual channels and communicate with the upper layers (FPCs, FPC Factory, Registry).

The third need was the design of the binding, a particular and crucial phase of the system. The general idea was to standardize a procedure that every type of device could easily use to connect to the system and begin working correctly and properly configured.

The fourth need was the design of the top layer, responsible for the ab-

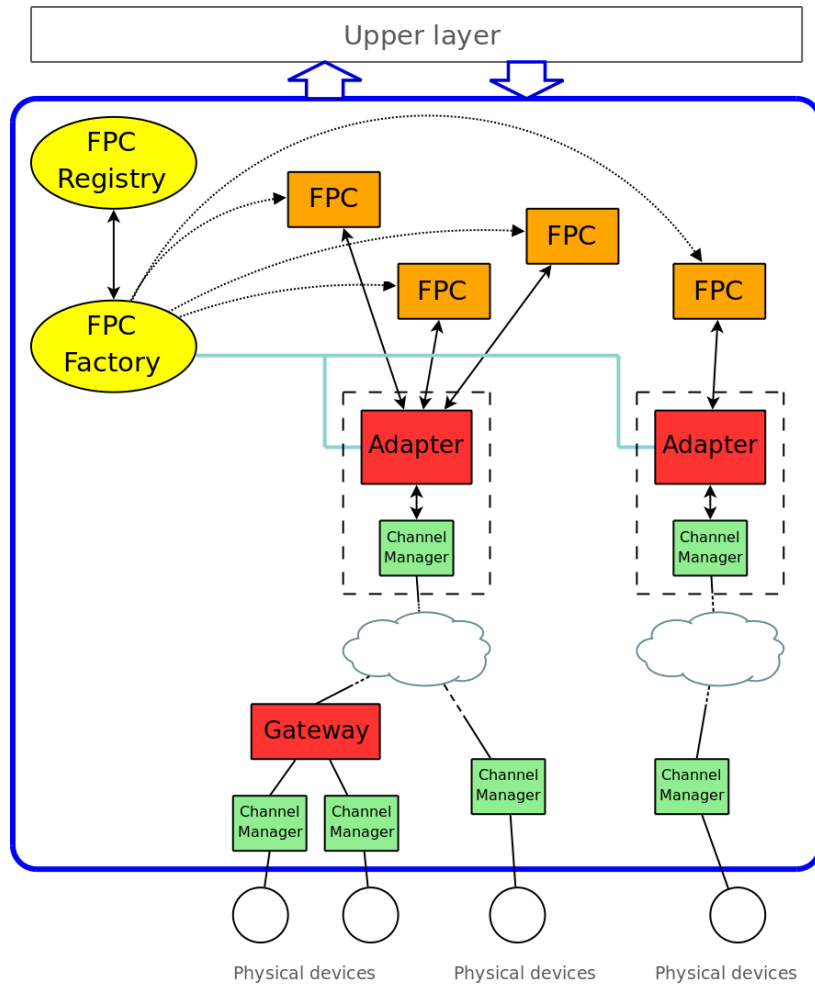


Figure 2.1: Part of the middleware addressed in the project.

straction of the devices (FPCs, FPC Factory and Registry).

While the first three needs have been studied, designed, implemented and tested in detail, these last three components were implemented with limited features, just for test purposes and to have a complete view of the functionality of the middleware at the end of the work. Nevertheless we spent some time thinking about the future behavior and requirements that these components should follow, in order to comply to the lower layer that we implemented. All these ideas will be listed as guidelines in Section 3.2.

While reaching the goals, we were driven by several requirements listed in the next section.

2.1 Requirements

Here we list the requirements that we outlined at the beginning of the project.

1. Different types of network can be connected to the system.
2. Several devices can be connected to a single hub that is the one connected to the system.
3. Devices can be heterogeneous, even in the same network.
4. Minimum programming effort is required to the device designers.
5. Communication can support both addressful and addressless messages. The reason behind this is that some channels support the addressing directly inside the messages (addressful), while others don't (addressless). In order to fulfill the requirement of heterogeneity, the system must accept both the situations.
6. Devices and FPCs can be changed at run time (plug & play fashion). The configuration of the network can change during the lifetime of the system and yet the system has to be able to reconfigure itself to accept the modifications and behave accordingly.
7. The logic can be split on different machines (many adapters and channels).

Chapter 3

Proposed architecture

The system we designed mainly consists of 7 major components:

1. Channel
2. Channel Manager
3. Gateway
4. Adapter
5. FPC
6. FPC Factory
7. Registry

We respected the general architecture presented in Figure 2.1.

The **Channel** is the component that abstracts a bidirectional communication channel. It can function over different kinds of physical networks. The unit of information that is exchanged is the **Frame**, that can be headerful (it supports source or destination address) or headerless (payload only).

The **Channel Manager** handles the Channels. Every instance of a Channel Manager masks a single type of physical technology. It exposes to the upper levels a simple interface of two methods: *read* and *write*. These methods are masked by pipes, to avoid blocking situations. For every Frame, the Channel Manager prepares a message with the payload of the Frame and the source address of the component who sent it; then, the message is written in the pipe connected to the upper level. The other way, when a component needs to send on the physical channel, it writes the payload and the destination address on the input pipe of the Channel Manager. To support a

possible shut down of the Channel, the Channel Manager should be able to re-create the connection.

On top of this last component there is the **Adapter**, whose purposes are:

1. To use the Channel Manager to connect a device to its relative FPC
2. To help in the binding of a device, by contacting the FPC Factory
3. To handle *Virtual Channels*

A Virtual Channel is the connection between a device and its relative FPC. It identifies a unique path in a particular network, but the Channel and the Channel Manager totally ignore this concept.

The **Gateways** are components deployed inside the network to perform the routing of the messages. They have one or more Channels going “up” (towards the FPCs) and coming from the “bottom” (from the devices). They keep a simple routing table needed for the translation of VCIs and network addresses. Upon creation, a Gateway receives a list of Channel Managers connected to it and a routing table with this structure:

- identifier of the source Channel Manager
- identifier of the destination Channel Manager
- destination address of the next hop towards the FPC Factory

This table is needed for the binding phase. Subsequently it keeps track of the already sent binding requests, waits for the acknowledge containing the VCI and builds a table for future routing of the messages of a device.

The **FPC Factory** is dedicated to the creation of FPCs. It is called by the Adapter upon receiving a binding request from a device. The Factory generates a new specific FPC based on the description of the device received from the Adapter. Then it links the FPC to the proper Adapter (to complete the Virtual Channel) and registers the pair (device,FPC) in the Registry. Right before the creation of the FPC, the Factory also checks if everything is correct by asking to the Registry.

The **Registry** keeps track of all the FPCs active in the system, along with their description and their attributes.

3.1 Differences with initial documentation

Initial documentation mainly consisted in the work of engineer Rota, as part of his imminent MSc thesis. The design of the project has proved to be reasonable and detailed. Considering the fact that we tried to follow initial guidelines as much as possible, the final differences are very limited. They're two to be precise.

1. The packet exchanged among Channels has an initial flag that precisely identifies the type of message.
2. The Frame can support any number of fields in future implementation, not only addresses, thanks to the adoption of a header.

3.2 Guidelines for future developments

We tried to address every aspect of the system architecture, at least for the part we were interested in. Consequently we can only give guidelines that merely continue our path of development.

1. The device descriptor needs to be carefully designed. For now it's supposed to be an XML file. Whatever its format, deep knowledge and analysis of all possible devices are a must for a correct definition of this file.
2. Related to the previous note, there is the need of a definition of a FPC for every kind of device. Each FPC will send custom payloads to communicate with the paired device.
3. We omitted details about the Registry, which is no doubt an important component. It has to be unique and to allow efficient search and classification of devices. Many good design patterns for the implementation of a registry of software components have been established.

Chapter 4

Software design

In this chapter we'll describe the implementation of Java classes and packages.

4.1 Class diagrams

4.1.1 Package `device.adapter`

First we'll give a general overview of the whole package (Figure 4.1).

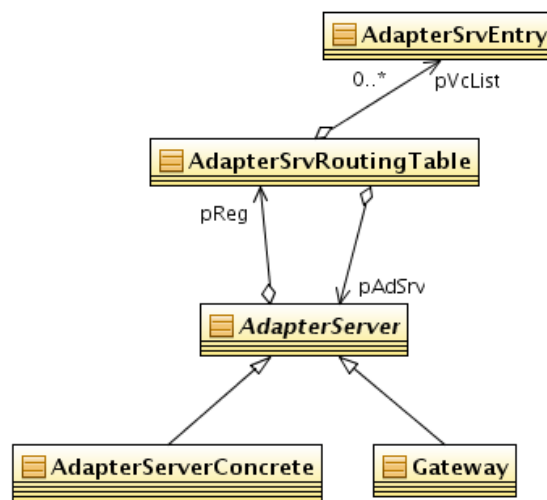


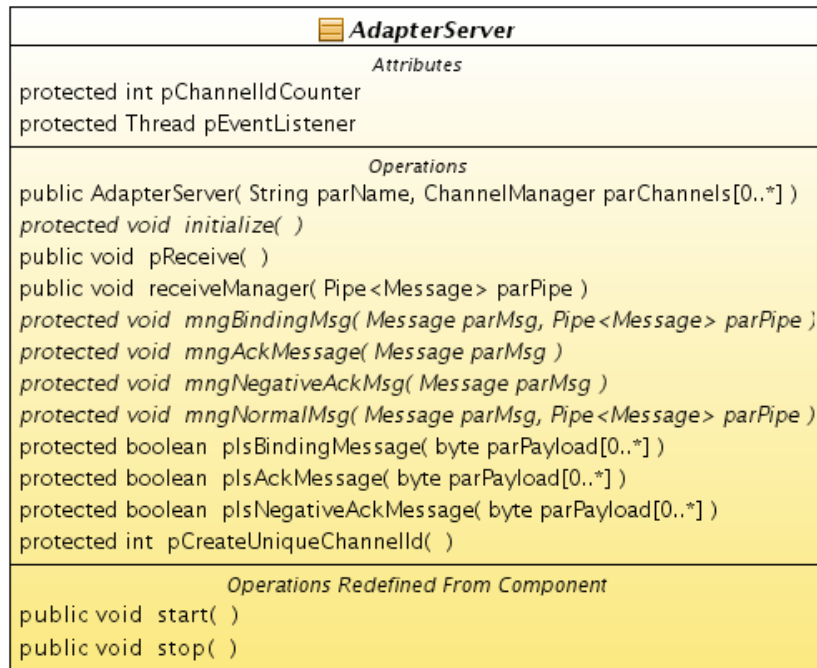
Figure 4.1: Structure of the package.

AdapterServer is an abstract class that is extended by two concrete


classes: the *AdapterServerConcrete* and the *Gateway*. The reason for this is to avoid duplicating the code common to both of them. In fact, since they operate at the same logical level of the communication, the core methods are equal.


Every *AdapterServer* keeps an *AdapterSrvRoutingTable* to handle the FPCs, their VCI and attached pipes. In the particular case of the *Gateway*, the address of the *Adapter* and the re-mapped BVCI are used, while pipes are not considered. The *AdapterSrvRoutingTable* maintains all this information by using a list of *AdapterSrvEntry*, which are basically tuples.

Here follow the detailed class diagram of all the classes of the package.





CHAPTER 4. Software design

 AdapterServerConcrete
<i>Attributes</i> <pre>protected int bindDomain = 2000000000 protected int maxQueueLength = 100000 private int pPipeCounter</pre>
<i>Operations</i> <pre>public AdapterServerConcrete(String parName, ChannelManager parChannels[0..*]) private pCreateNewPipes() private boolean pBindNewDevice(Message parMsg, Pipe<Message> parChannelPipeIn, Pipe<Message> parChannelPipeOut) protected void mngNormalMsg(Message parMsg, Pipe parPipe) public void addNewChannel(Channel c) private Pipe pGetPipeOutFromPipeIn(Pipe parPipeIn) public Pipe<AdapterFpcMessage> getFactPipeIn() public Pipe<AdapterFpcMessage> getFactPipeOut()</pre>
<i>Operations Redefined From AdapterServer</i> <pre>protected void mngAckMessage(Message parMsg) protected void mngBindingMsg(Message parMsg, Pipe<Message> parPipe) protected void mngNegativeAckMsg(Message parMsg) protected void initialize()</pre>
<i>Operations Redefined From Component</i> <pre>public void start() public void stop()</pre>


 Gateway
<i>Attributes</i> <pre>private int pStaticRoutesSrcChannels[0..*] private int pStaticRoutesDstChannels[0..*]</pre>
<i>Operations</i> <pre>public Gateway(ChannelManager parChannels[0..*]) public void setStaticRoutes(int parStaticRoutesSrcChannels[0..*], int parStaticRoutesDstChannels[0..*], FrameAddress dstAddress[0..*]) protected void mngNormalMsg(Message parMsg, Pipe parPipe) public void addNewChannel(ChannelManager parC) public void removeRoute(int parSrc, int parDst) public void addRoute(int parSrc, int parDst, FrameAddress parDstAddress) private Pipe getPipeOutFromPipeIn(Pipe parPipe) public int getIndexFromPipeIn(Pipe<Message> parPipe)</pre>
<i>Operations Redefined From AdapterServer</i> <pre>protected void initialize() protected void mngAckMessage(Message parMsg) protected void mngBindingMsg(Message parMsg, Pipe<Message> parPipe) protected void mngNegativeAckMsg(Message parMsg)</pre>


CHAPTER 4. Software design

 AdapterSrvRoutingTable
<i>Attributes</i>
<i>Operations</i>
<pre> public AdapterSrvRoutingTable(AdapterServer parAdSrv) public boolean addEntry(int parBvci, Pipe<Message> parUpperPipeIn, Pipe<Message> parUpperPipeOut, Pipe<Message> parLowerPipe public boolean addEntry(int parBvci, Pipe<Message> parUpperPipeIn, Pipe<Message> parUpperPipeOut, Pipe<Message> parLowerPipe public int setNewVci(int parBvci, FrameAddress parAddress) public int getVciFromUpperPipeIn(Pipe<Message> parPipe) public int getVciFromUpperPipeOut(Pipe<Message> parPipe) public int getUpperPipesFromVci(int parIdentifier) public int getPipesOutFromVci(int parIdentifier) public int getPipesInFromVci(int parIdentifier) public int getUpperPipesFromBvciAndDeviceAddress(int parIdentifier, FrameAddress parAddress) public int getVciFromLowerPipeIn(Pipe<Message> parPipe) public int getVciFromLowerPipeOut(Pipe<Message> parPipe) public int getLowerPipesFromVci(int parIdentifier) public int getLowerPipesFromBvciAndDeviceAddress(int parIdentifier, FrameAddress parAddress) public boolean removeEntryByBvci(int parBvci) public boolean removeEntryByVci(int parVci) private AdapterSrvEntry pGetElFromBvciAndDeviceAddress(int parIdentifier, FrameAddress parAddress) private AdapterSrvEntry pGetElFromVci(int parIdentifier) public FrameAddress getDeviceAddressFromVci(int parVci) public AdapterSrvEntry[] getAllEntries() public void setVciFromBvciAndDeviceAddress(int parVci, int parBvci, FrameAddress parDeviceAddr) public void setVciFromBvciRemappedAndAdapterAddress(int parVci, int parBvci, FrameAddress parAdapterAddr) public boolean removeEntryByBvciAndDeviceAddress(int parBvci, FrameAddress parDeviceAddr) public void setRemappedBvciFromBvciAndDeviceAddress(int parRemappedBvci, int parBvci, FrameAddress parDeviceAddr) public int getBvciFromVci(int parVci) public FrameAddress getDeviceAddressFromBvciRemappedAndAdapterAddress(int parRemappedBvci, FrameAddress parAdapterAddr) public int getBvciFromBvciRemappedAndAdapterAddress(int parRemappedBvci, FrameAddress parAdapterAddr) public FrameAddress getAdapterAddressFromVci(int parVci) public void Unnamed() </pre>
 AdapterSrvEntry
<i>Attributes</i>
<pre> private int pBvci private int pRemappedBvci private int pVci </pre>
<i>Operations</i>
<pre> public AdapterSrvEntry(int parBvci, Pipe<Message> parUpperPipeIn, Pipe<Message> parUpperPipeOut, public AdapterSrvEntry(int parBvci, Pipe<Message> parUpperPipeIn, Pipe<Message> parUpperPipeOut, public int getBvci() public void setBvci(int bvci) public FrameAddress getDeviceAddress() public void setDeviceAddress(FrameAddress deviceAddress) public Pipe<Message> getLowerPipeIn() public void setLowerPipeIn(Pipe<Message> lowerPipeIn) public Pipe<Message> getLowerPipeOut() public void setLowerPipeOut(Pipe<Message> lowerPipeOut) public int getRemappedBvci() public void setRemappedBvci(int remappedBvci) public Pipe<Message> getUpperPipeIn() public void setUpperPipeIn(Pipe<Message> upperPipeIn) public Pipe<Message> getUpperPipeOut() public void setUpperPipeOut(Pipe<Message> upperPipeOut) public int getVci() public void setVci(int vci) public FrameAddress getAdapterAddress() public void setAdapterAddress(FrameAddress adapterAddress) </pre>

4.1.2 Package device.channel

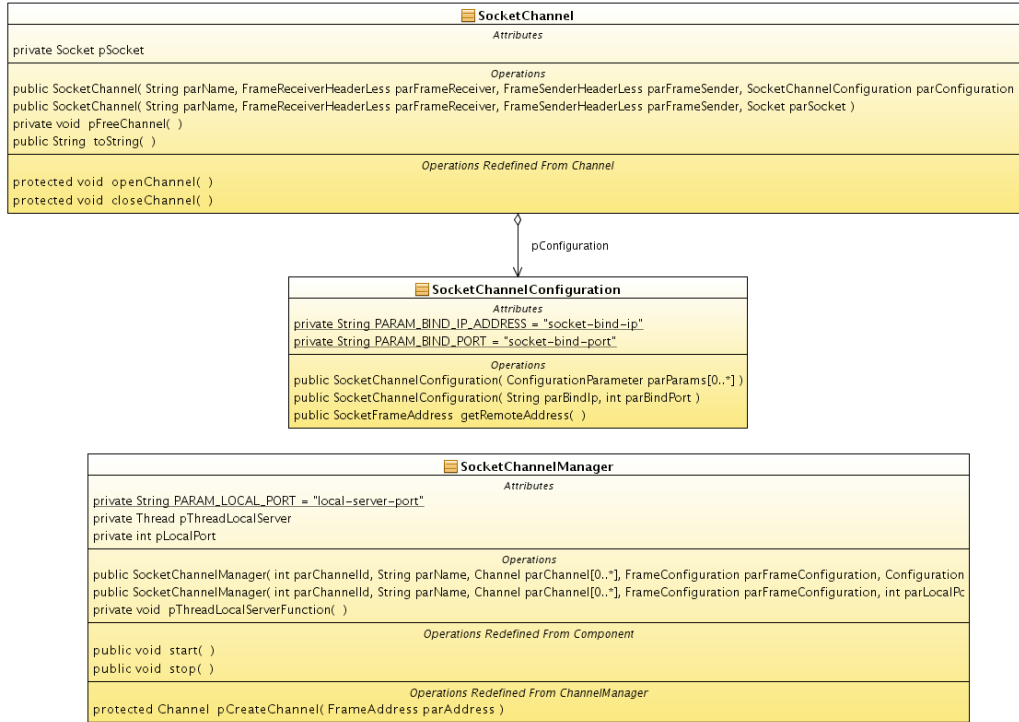
The abstract class *ChannelManager* provides the communication interfaces towards the upper levels, by the input and output pipes. The configurator instantiates each concrete *ChannelManager* by reading from the configuration XML of the single node. The *ChannelManager* objects then instantiate the concrete objects of the abstract class *Channel*, based on the type of underlying technology handled by the *ChannelManager*.

 Channel
<i>Attributes</i>
<pre>private OutputStream pChannelOutputStream private InputStream pChannelInputStream private boolean pEventBasedInput private Thread pThreadSender private Thread pThreadReceiver</pre>
<i>Operations</i>
<pre>private Channel(String parName, FrameReceiver parFrameReceiver, FrameSender parFrameSender, FrameAddress parLocalAddress) public Channel(String parName, FrameReceiverHeaderLess parFrameReceiver, FrameSenderHeaderLess parFrameSender, FrameAddress parLocalAddress) public Channel(String parName, FrameReceiverHeaderFul parFrameReceiver, FrameSenderHeaderFul parFrameSender, FrameAddress parLocalAddress) protected void setEventBasedInput(boolean parEventBasedInput) protected boolean getEventBasedInput() protected void setInputStream(InputStream parInputStream) protected InputStream getInputStream() protected void setOutputStream(OutputStream parOutputStream) protected OutputStream getOutputStream() public void setRemoteAddress(FrameAddress parRemoteAddress) public FrameAddress getRemoteAddress() public Pipe<AdapterChannelMessage> getInputPipe() public Pipe<AdapterChannelMessage> getOutputPipe() protected void openChannel() protected void closeChannel() private void pSenderThreadFunction() private void pReceiverThreadFunction() protected void receive() private void pReceive(boolean parBlockingMode)</pre>
<i>Operations Redefined From Component</i>
<pre>public void start() public void stop()</pre>

 ChannelManager
<i>Attributes</i>
<pre>private int pChannelId private Thread pThreadChannelRoutingOutput private Thread pThreadChannelRoutingInput</pre>
<i>Operations</i>
<pre>public ChannelManager(int parChannelId, String parName, Channel parChannel[0..*], FrameConfiguration parFrameConfiguration) public Pipe<AdapterChannelMessage> getInputPipe() public Pipe<AdapterChannelMessage> getOutputPipe() public int getChannelId() private void pThreadChannelRoutingOutputFunction() private void pThreadChannelRoutingInputFunction() protected Channel pCreateChannel(FrameAddress parAddress) protected void pRegisterChannel(Channel parChannel, FrameAddress parRemoteAddress) public String toString()</pre>
<i>Operations Redefined From Component</i>
<pre>public void start() public void stop()</pre>

4.1.3 Package device.channel.concrete.socket

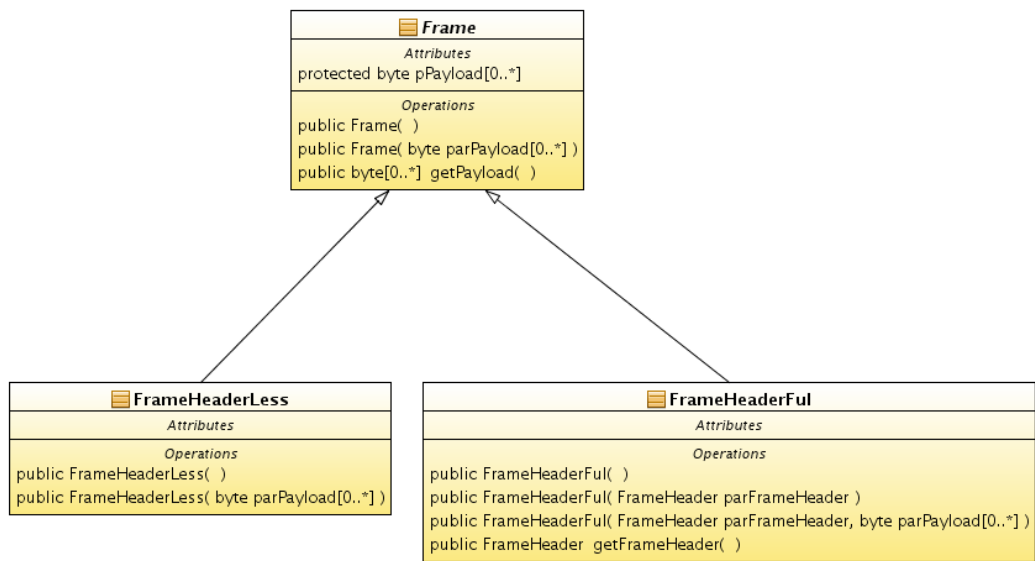
The package contains the implementations of a concrete *ChannelManager* and its related channels, based on the TCP/IP socket.



4.1.4 Package `device.channel.frame`

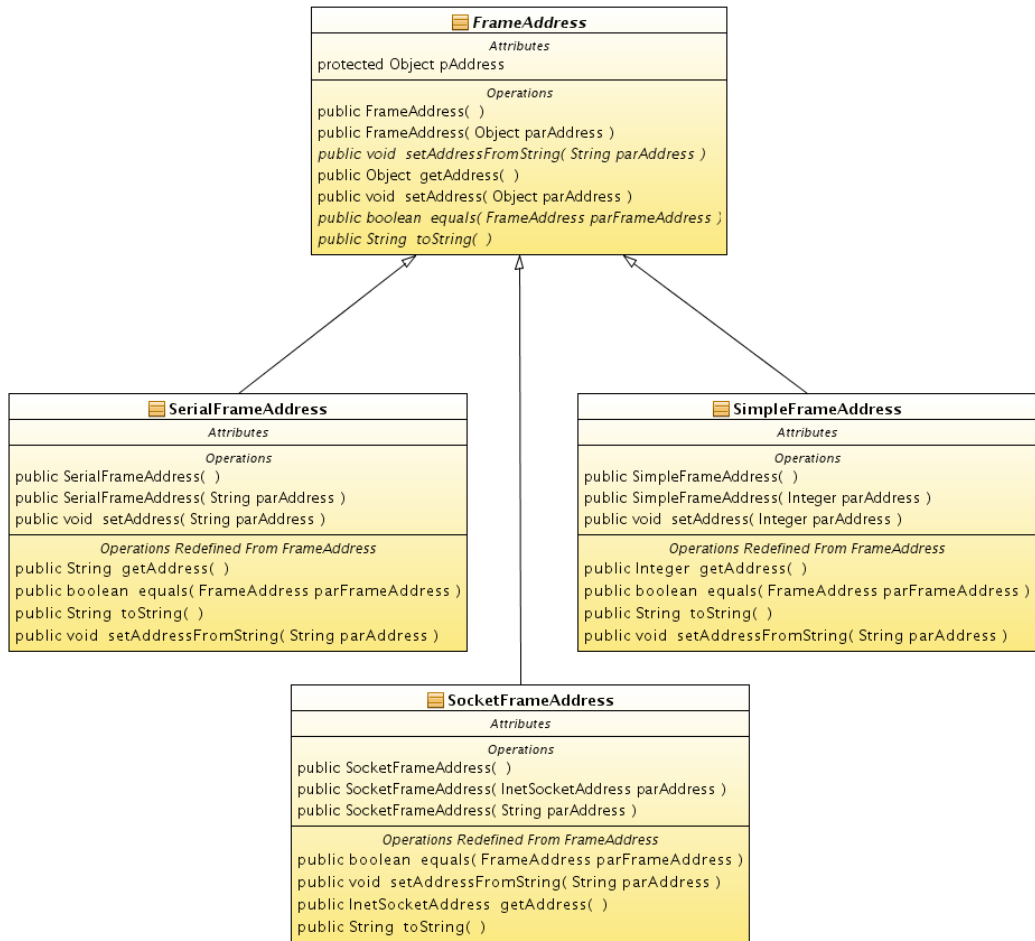
This package abstracts the messages sent through a physical channel. Two kinds of Frame are supported:

1. **headerless**: for channels with addressing support or that do not require further fields in the Frame.
2. **headerful**: for channels without native addressing support, or to allow future expansions in the structure of the Frame.



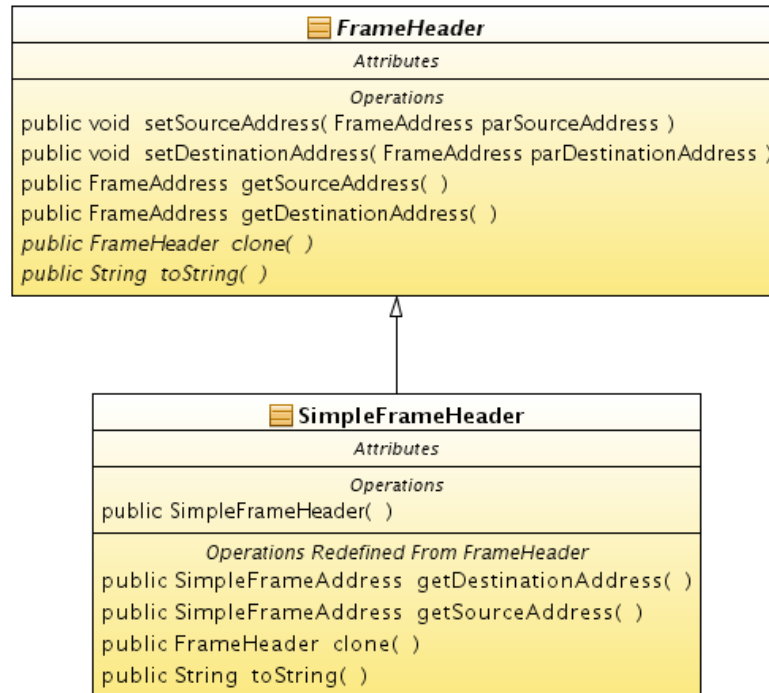
4.1.5 Packages `device.channel.frame.address` and `device.channel.frame.address.concrete`

The package contains the abstract class *FrameAddress*, representing the address of a frame, and various concrete implementations for different types of channels.



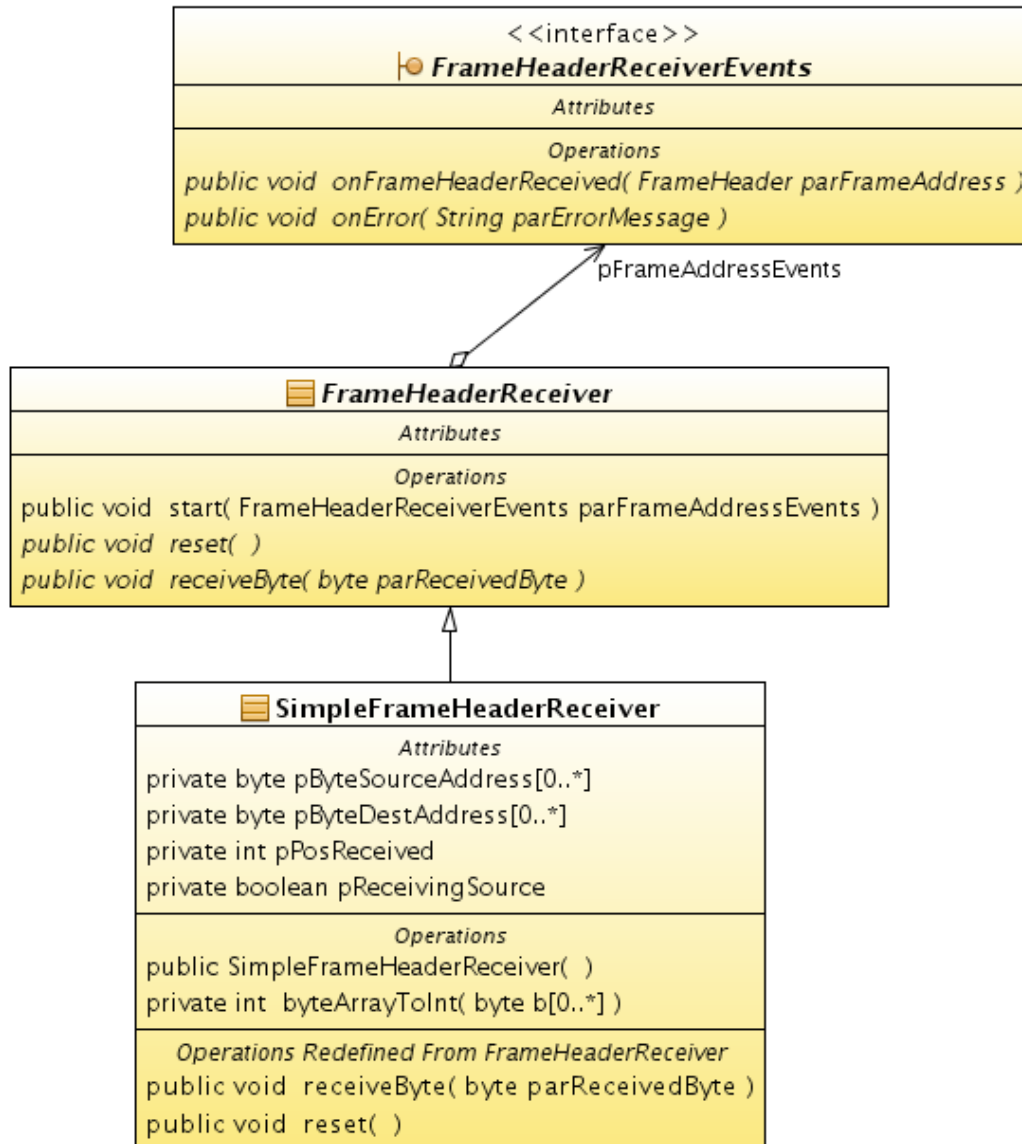
4.1.6 Package `device.channel.frame.header` and `device.channel.frame.header.concrete.simple`

The package contains the abstract class *FrameHeader*, for the management of the headers of the headerful frames, and an example of concrete implementation.



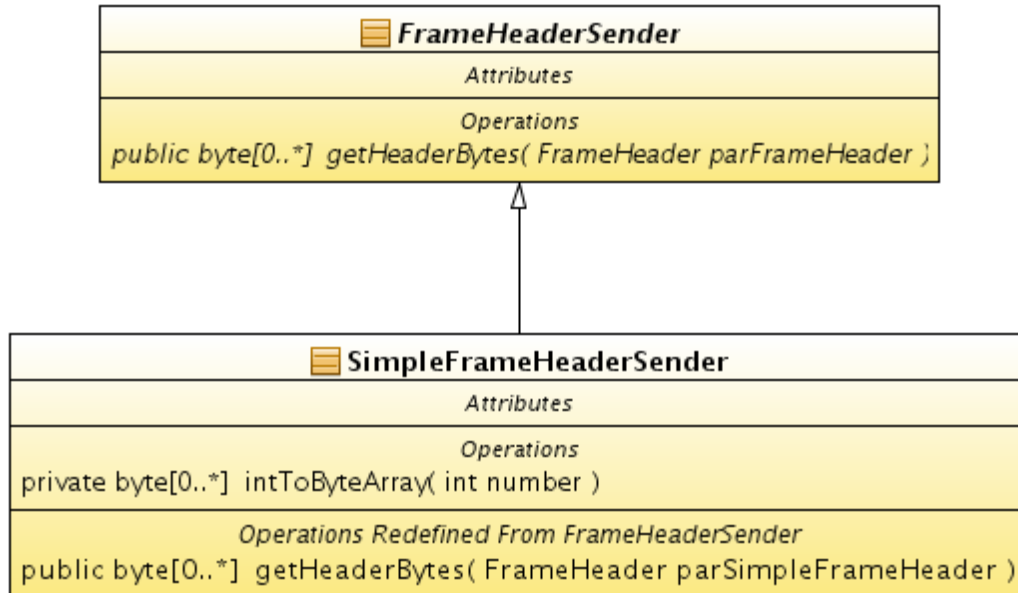
4.1.7 Package device.channel.frame.receiver

Class to “decode” a particular header from raw data received through the physical channel.



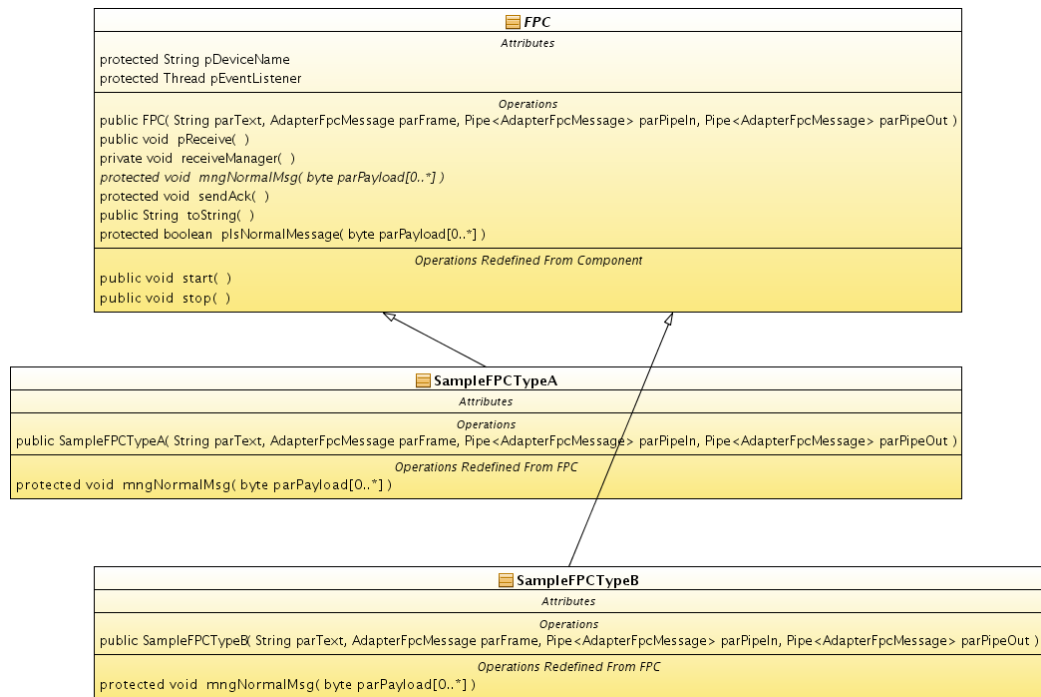
4.1.8 Package `device.channel.frame.sender`

Class to “encode” a particular header on the underlying physical channel.




4.1.9 Package device.fpc

This is a very simple implementation of a FPC, used to complete the system. It's a stub for future developments. Two concrete realizations of the abstract *Fpc* class are provided, mainly for test purposes.



4.1.10 Package `device.fpcfactory`

Similar to the *fpc* package, the *FpcFactory* class is a bare bone implementation. It handles the creation of new *Fpc* objects. The class also contains some logic that in the designed architecture should be provided by the Registry: this was necessary to build a functioning system.

 FpcFactory
<i>Attributes</i>
<code>protected String name = "FPC Factory"</code> <code>private Thread pEventListener</code>
<i>Operations</i>
<code>public FpcFactory(AdapterServerConcrete parAdapter)</code> <code>public void pReceive()</code> <code>public void receiveManager()</code> <code>private boolean plsEqual(byte array1[0..*], byte array2[0..*])</code> <code>private boolean makeFpc(String parDesc, AdapterFpcMessage parMsg, Pipe parPipeIn, Pipe parPipeOut, int parDeviceType)</code> <code>public Waiter getFactWaiter()</code>
<i>Operations Redefined From Component</i>
<code>public void start()</code> <code>public void stop()</code>

4.1.11 Package configurator


The *ConfigurationManager* basically takes care of the initialization of all the components that need the retrieving of configuration parameters from the XML descriptor. It provides public methods to obtain the components that are instantiated after the start of the *ConfigurationManager*.

ConfigurationManager
<i>Attributes</i> <pre>private String TAG_PARAM = "param" private String TAG_PARAM_NAME = "name" private String TAG_CHANNEL_MANAGER = "channel-manager" private String TAG_CHANNEL_MANAGER_ID = "channel-manager-id" private String TAG_CHANNEL_MANAGER_NAME = "channel-manager-name" private String TAG_CHANNEL_MANAGER_TYPE = "channel-manager-type" private String TAG_CHANNEL_MANAGER_CONFIG = "channel-manager-config" private String TAG_CHANNEL = "channel" private String TAG_CHANNEL_NAME = "channel-name" private String TAG_CHANNEL_CONFIG = "channel-config" private String TAG_CHANNEL_FRAME_CONFIG = "frame-config" private String TAG_CHANNEL_FRAME_HEADER_CONFIG = "frame-header-config" private String TAG_CHANNEL_FRAME_HEADER_TYPE = "frame-header-type" private String TAG_CHANNEL_FRAME_HEADER_ADDRESS = "frame-local-address" private String TAG_GATEWAY = "gateway" private String TAG_GATEWAY_ROUTE_BINDING = "route-binding" private String TAG_GATEWAY_ROUTE_BINDING_SRC = "route-binding-src" private String TAG_GATEWAY_ROUTE_BINDING_DST = "route-binding-dst" private String TAG_GATEWAY_ROUTE_BINDING_DST_ADDRESS = "route-binding-dst-address" private String TYPE_FRAME_HEADER_SIMPLE = "simple" private String TYPE_CHANNEL_SOCKET = "socket" private String TYPE_CHANNEL_SERIAL = "serial" private String pConfigFilePath private Document pDoc</pre>
<i>Operations</i> <pre>public ConfigurationManager(String parConfigFile) public ChannelManager[0..*] getChannels() public Gateway getGateway() private boolean pParseConfiguration() private void pParseChannelConfig() private void pParseGatewayConfig() private FrameHeaderConfig pSelectFrameHeader(String parStr, String parLocalAddressStr) private ConfigurationParameter[0..*] pExtractParameter(Element parElement)</pre>
<i>Operations Redefined From Component</i> <pre>public void start()</pre>

4.1.12 Package utils.messages

The package contains the messages exchanged between logical levels of the same node. *AdapterChannelMessage* keeps the info needed to send/receive data to/from a Gateway or an Adapter from/to the underlying physical channel. *MsgUtility* contains several static methods used by other classes.



 MsgUtility
<p><i>Attributes</i></p> <pre> public int flagAck = 0 public int flagBinding = 1 public int flagNegativeAck = 2 public int flagOthers = 3 private int startFlagOffset = 0 private int startBvciOffset = 4 private int startVciOffsetAckMessage = 8 private int startVciOffset = 4 protected int bindDomain = 2000000000 public int channelIdLength = 4 public int flagLength = 4 </pre>
<p><i>Operations</i></p> <pre> public byte[0..*] intToByteArray(int parNumber) public int getFlag(byte parPayload[0..*]) public int getVciFromAckMessage(byte parPayload[0..*]) public int getBvci(byte parPayload[0..*]) public int getVciNormalMessage(byte parPayload[0..*]) public int byteArrayToInt(byte b[0..*]) public int getIntFromPayload(byte payload[0..*], int start, int end) public byte[0..*] createAckPayload(int parBvci, int parVci) public byte[0..*] createNegativeAckPayload(int parBvci) public byte[0..*] modifiedBvciBindingMessage(byte parPayload[0..*], int parBvci, int parRemappedBvci) public byte[0..*] getNormalMsgContent(byte parPayload[0..*]) </pre>

4.2 Guidelines for future developments

The three components at the top of our system (FPC Factory, FPC, Registry) needs imminent development, since we just provided a very limited but working implementation of them.

The **FPC** class can be designed only by knowing how devices should work and what kind of configuration they can have. The important part here is to remain as general as possible, in order to support all the types of devices. Yet every FPC must be properly configured in detail, to respond exactly as a device expects. The FPC should have basically three families of methods:

1. a constructor called by the Factory that receives the configuration of the object
2. methods to communicate with the paired device, to request and to specify a particular behavior
3. methods exposed to the upper levels, in order to handle queries coming from the parser

The **FPC Factory** class can be static, because it receives binding requests from the Adapter, completed with all needed information, and it uses the Registry for every check that it may need. The core of the implementation deals with the parsing of the descriptor of the device (probably an XML file) and the consequent extraction of the features of the FPC to generate.

The **Registry** class has to keep a table of all the active FPCs, along with the descriptor of the device, its logical address and possibly the status of the device. It is responsible for the creation of logical addresses, in the case a device doesn't have a static one.

Another part to develop deals with the Channel. Only the class specific for the TCP/IP socket is ready. Other types must be provided, along with their configuration directives.

Chapter 5

Testing

In this chapter we'll describe the tests we performed on the implemented software. To demonstrate the correctness of the implementation and the respect of the requirements, we outlined three significant scenarios.

All tests are in separate folder with respect to the source of the system. The output of the execution is handled by the Logger class. All components have their relative configuration XML files that are loaded by the Configuration Manager upon initialization.

Since tests run on a local machine with the same address (127.0.0.1), at the TCP/IP level we used different ports to simulate different addresses. Note that by executing the test on another machine port numbers could change, as they're assigned from the Operating System.

For the second and third tests we're going to show the detailed operations occurring during execution. The tables that are going to be used are explained here.

Phase 1: start of channels among devices

The channels created during this phase are listed in the tables below. The name of the node is shown in the first row of each table, while the other rows contain the active channels.

Local port	Local TCP port, to which a channel is connected
Remote port	Remote TCP port, to which a channel is connected
Channel Manager ID	Identifier of the Channel Manager that manages a channel

Phase 2: binding of devices

Adapter address	Address of the adapter to which we want to send a message
Device address	Physical address of the device
Bvci	Binding virtual channel id
Remapped bvci	Remapped bvci
Vci	Id of the final virtual channel
UpipIn	Reading pipe from the upper level
UpipOut	Writing Pipe to the upper level
LpipIn	Reading pipe from the lower level
LpipOut	Writing Pipe to the lower level
CMPipIn	Reading pipe from the channel manager
CMPipOut	Writing Pipe to the channel manager

5.1 The Socket Channel

This test can be found in the package `org.dei.perla.device.channel`. The goal of the test is to show the correct functioning of the low level communication between two nodes using TCP/IP sockets.

The simulated network is composed of two devices (B, C) and a node that acts as a server to which the devices connect. The devices:

1. create a connection towards the server
2. send 5 messages (with random payload) each, towards the server
3. close the connection

The server simply receives the messages. Each node uses the Channel Manager to create the Channels.

5.2 A complete network

This test can be found in the package `org.dei.perla.device.network`. The goal is to demonstrate the behavior of a complete network, with emphasis on the binding phase. The network is composed of:

- 5 devices (1a, 1b, 2a, 2b, 3)
- a Gateway
- a server machine that holds an Adapter Server and a FPC Factory

The structure of the network is shown in Figure 5.1.

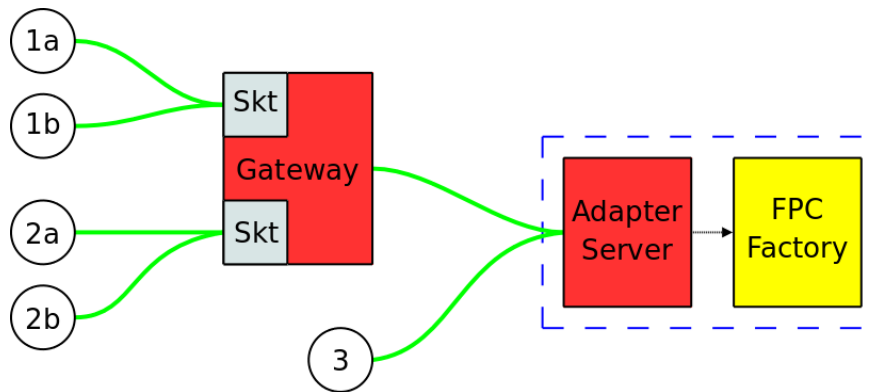


Figure 5.1: Structure of the simulated network.

The test shows how the gateway works in order to mask the underlying network: this can be seen by analyzing the payload of the exchanged messages. Each device:

1. sends a binding message towards the Factory
2. waits for the acknowledgment
3. sends a normal message towards its relative FPC
4. waits for the reply
5. terminates

Adapter Servers located in every server machine handle binding requests by routing messages from lower levels to the FPC factory, which instantiates the FPC. After the FPC has been created, it is directly connected to the Adapter Server and it will be its own responsibility to communicate with the

device. At the beginning, it sends the acknowledgment message to signal the correct termination of the binding procedure. Then it communicates with the device, by invoking appropriate device features.

The server machine handles the messages through the Adapter Server, which in turn communicates with the Factory for the creation of the FPC, given a binding request. Then it's the newly created FPC that replies back to its relative device. It sends the acknowledgment first and sends an answer to the normal message after.

The Gateway translates the addresses from the devices to the server machine and back and forth, by using its routing tables:

1. receives a binding message from a device, with the BVCI
2. re-maps the BVCI and saves the route from the device to the next hop
3. receives the acknowledgment message with the VCI
4. saves VCI and sends the acknowledgment message to the device, including the old BVCI in the message too
5. receives a normal message from a device and sends it along the correct path
6. receives a normal message and sends it to the addressee device

Each device upon initialization receives the binding address, which is the one belonging to the next hop towards the FPC Factory.

Phase 1

Gateway connection on port 7000 of FPC Factory

GATEWAY

Local port	Remote port	Channel Manager ID	Notes
42792	7000	1	Gateway connection - FPC Factory

FPC Factory

Local port	Remote port	Channel Manager ID	Notes
7000	42792	1	Gateway connection - FPC Factory

Sensor 1a connection on port 4000 of Gateway

SENSOR 1a

Local port	Remote port	Channel Manager ID	Notes
44395	4000	1	Sensor 1a connection - Gateway

GATEWAY

Local port	Remote port	Channel Manager ID	Notes
42792	7000	1	Gateway connection - FPC Factory
4000	44395	2	Sensor 1a connection - Gateway

Sensor 2a connection on port 4000 of Gateway

SENSOR 2a

Local port	Remote port	Channel Manager ID	Notes
44397	4000	1	Sensor 2a connection - Gateway

GATEWAY

Local port	Remote port	Channel Manager ID	Notes
42792	7000	1	Gateway connection - FPC Factory
4000	44395	2	Sensor 1a connection - Gateway
4000	44397	2	Sensor 2a connection - Gateway

Sensor 1b connection on port 5000 of Gateway

SENSOR 1b

Local port	Remote port	Channel Manager ID	Notes
34445	5000	1	Sensor 1b connection - Gateway

GATEWAY

Local port	Remote port	Channel Manager ID	Notes
42792	7000	1	Gateway connection - FPC Factory
4000	44395	2	Sensor 1a connection - Gateway
4000	44397	2	Sensor 2a connection - Gateway
5000	34445	3	Sensor 1b connection - Gateway

Sensor 2b connection on port 5000 of FPC Factory

SENSOR 2b

Local port	Remote port	Channel Manager ID	Notes
34447	5000	1	Sensor 2b connection - Gateway

GATEWAY

Local port	Remote port	Channel Manager ID	Notes
42792	7000	1	Gateway connection - FPC Factory
4000	44395	2	Sensor 1a connection - Gateway
4000	44397	2	Sensor 2a connection - Gateway
5000	34445	3	Sensor 1b connection - Gateway
5000	34447	3	Sensor 2b connection - Gateway

Sensor 3 connection on port 7000 of Gateway

SENSOR 3

Local port	Remote port	Channel Manager ID	Notes
44398	7000	1	Sensor 3 connection - FPC Factory

FPC Factory

Local port	Remote port	Channel Manager ID	Notes
7000	42792	1	Gateway connection - FPC Factory
7000	44398	1	Sensor 3 connection - FPC Factory

Phase 2

Gateway routing tables:

FROM	TO
1	3
2	3

The binding procedure for nodes 1a, 2a, 1b, 2b is showed below. Details are given for node 1b only, other nodes act in the same way.

1. Node 1b sends the binding message that arrives at the gateway that handles it:

BINDING FLAG + BVCI + XML DESCRIPTOR , device address

1 + BVCI_1b + <xml descriptor> ... </xml descriptor>, address1b

Gateway:

Adapter address	3.adapterAddress
Device address	address1b
Bvci	BVCI_1b
Remapped Bvci	2333
Vci	0
UpipeIn	3.pipein
UpipeOut	3.pipeout
LpipeIn	2.pipein
LpipeOut	2.pipeout

We're using the second static route.

2. A new ack message is created, swapping the arrived bvci with the remapped bvci:

1 + 2333 + <xml descriptor> ... </xml descriptor>, address1b

3. The message arrives at the adapter server and its final tables will be like this:

AdapterServer:

Adapter address	null
Device address	address1b
Bvci	2333
Remapped Bvci	null
Vci	0
UpipeIn	FPCpipeIn
UpipeOut	FPCpipeOut
LpipeIn	CMpipeIn
LpipeOut	CMpipeOut

4. After being created, the new FPC, with the 2 new pipes FPCpipeIn e FPCpipeOut, sends an ack message to the adapter server, which creates the new VCI and updates its tables in this way:

AdapterServer:

Adapter address	null
Device address	address1b
Bvci	2333
Remapped Bvci	null
Vci	1
UpipeIn	FPCpipeIn
UpipeOut	FPCpipeOut
LpipeIn	CMpipeIn
LpipeOut	CMpipeOut

5. The ack message is sent to the lower level through the channel manager:

Ack flag + BVCI + VCI

0 + 2333 + 1

6. The gateway handles the ack message and updates its routing tables:

Gateway1:

Adapter address	3.adapterAddress
Device address	address1b
Bvci	BVCI_1b
Remapped Bvci	2333
Vci	1
UpipeIn	3.pipein
UpipeOut	3.pipeout
LpipeIn	2.pipein
LpipeOut	2.pipeout

7. It creates a new ack message and sends it to the lower level, swapping the arrived bvci with the bvci in its table:

Ack flag + BVCI + VCI

0 + BVCI_1b + 1

Binding of node 3:

BINDING FLAG + BVCI + XML DESCRIPTOR, device address

1 + BVCL3 + <xml descriptor> ...</xml descriptor>, address3

8. The message arrives at the adapter server and its final tables will be like this:

AdapterServer:

Adapter address	null	null
Device address	address1b	Address3
Bvci	2333	BVCL3
Remapped Bvci	null	null
Vci	1	0
UpipeIn	FPCpipeIn	FPC2pipeIn
UpipeOut	FPCpipeOut	FPC2pipeOut
LpipeIn	CMpipeIn	CMpipeIn
LpipeOut	CMpipeOut	CMpipeOut

9. After being created, the new FPC, with the 2 new pipes FPC2pipeIn e FPC2pipeOut, sends an ack message to the adapter server that creates the new VCI and updates its tables in this way:

AdapterServer:

Adapter address	null	null
Device address	address1b	Address3
Bvci	2333	BVCL3
Remapped Bvci	null	null
Vci	1	2
UpipeIn	FPCpipeIn	FPC2pipeIn
UpipeOut	FPCpipeOut	FPC2pipeOut
LpipeIn	CMpipeIn	CMpipeIn
LpipeOut	CMpipeOut	CMpipeOut

10. The ack message is sent to the lower level (node 3 directly) through the channel manager:

Ack flag + BVCI + VCI

0 + BVCL3 + 1

5.3 Two Gateways in waterfall fashion

The last test aims at proving the correctness of the Gateway component, even when two of them are deployed in series. It can be found in the package `org.dei.perla.device.multigateway`. The network is composed of:

- 2 devices (1a and 1b)
- 2 Gateways
- a server machine that holds an Adapter Server and a FPC Factory

The structure of the network is shown in Figure 5.2.

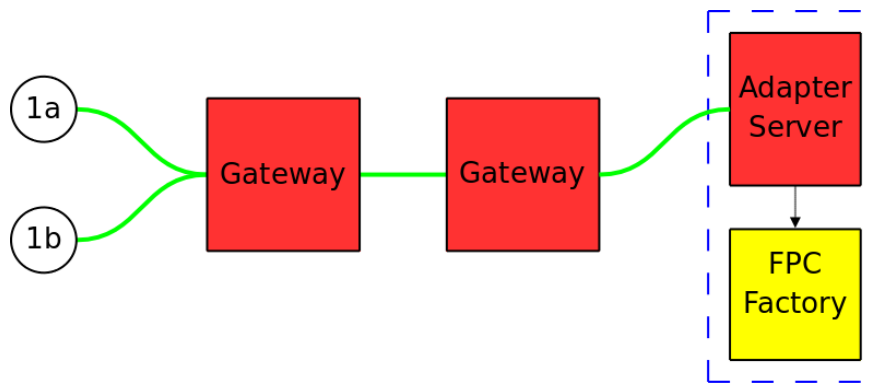


Figure 5.2: Structure of the simulated network.

The performed actions are similar to the previous test. The two devices simply send a binding message towards the gateways and wait for the acknowledgment. The Adapter Server, FPC Factory and FPCs behave according to the same binding routine: the Adapter Server communicates with the Factory for the creation of the FPC. The newly created FPC replies back to its relative device with an acknowledgment. The actions of the Gateways can be seen by analyzing the payload of the exchanged messages. This time the second Gateway (the one nearer to the Factory) simply forwards the messages of the first one.

Phase 1

Gateway 2 connection on port 8000 of FPC Factory

GATEWAY 2

Local port	Remote port	Channel Manager ID	Notes
57882	8000	1	Gateway2 connection - FPC Factory

FPC Factory

Local port	Remote port	Channel Manager ID	Notes
8000	57882	1	Gateway2 connection - FPC Factory

Gateway 1 connection on port 6000 of Gateway 2

GATEWAY 1

Local port	Remote port	Channel Manager ID	Notes
35231	6000	1	Gateway1 connection - Gateway2

GATEWAY 2

Local port	Remote port	Channel Manager ID	Notes
57882	8000	1	Gateway2 connection - FPC Factory
6000	35231	2	Gateway1 connection- Gateway2

Sensor 1 connection to Gateway 1

SENSOR 1

Local port	Remote port	Channel Manager ID	Notes
35233	4000	1	Sensor1 connection- Gateway1

GATEWAY 1

Local port	Remote port	Channel Manager ID	Notes
35231	6000	1	Gateway1 connection - Gateway2
4000	35233	2	Sensor1 connection- Gateway1

Sensor 2 connection to Gateway 1

SENSOR 2

Local port	Remote port	Channel Manager ID	Notes
35234	4000	1	Sensor2 connection-Gateway1

GATEWAY 1

Local port	Remote port	Channel Manager ID	Notes
35231	6000	1	Gateway1 connection-Gateway2
4000	35233	2	Sensor1 connection-Gateway1
4000	35234	2	Sensor2 connection-Gateway1

Phase 2

Static routes gateway 1 and gateway 2:

FROM	TO
1	2

1. The binding message arrives to gateway 1, with this format:

BINDING FLAG + BVCI + XML DESCRIPTOR, device address

1 + BVCI.1a + <xml descriptor> ... </xml descriptor>, address1a

Gateway1:

Adapter address	2.adapterAddress
Device address	address1a
Bvci	BVCI_1a
Remapped Bvci	34
Vci	0
UpipeIn	2.pipein
UpipeOut	2.pipeout
LpipeIn	1.pipein
LpipeOut	1.pipeout

Where i.e. 2.pipeIn is the reading pipe of the channel manager with local id 2 (the same id of the static routes).

2. A new binding message is created, with, instead of the arrived BVCI, the remapped BVCI:

1 + 34 + <xml descriptor> ... </xml descriptor>, address1a

3. Gateway 2 receives the message and it handles it:

Gateway2:

Adapter address	2.adapterAddress
Device address	address1a
Bvci	34
Remapped Bvci	56
Vci	0
UpipeIn	2.pipein
UpipeOut	2.pipeout
LpipeIn	1.pipein
LpipeOut	1.pipeout

4. A new binding message is created, with, instead of the arrived BVCI, the remapped BVCI:

1 + 56 + <xml descriptor> ... </xml descriptor>, address1a

5. The message arrives to the adapter server and its final tables will be like this:

AdapterServer:

Adapter address	null
Device address	address1a
Bvci	56
Remapped Bvci	null
Vci	0
UpipeIn	FPCpipeIn
UpipeOut	FPCpipeOut
LpipeIn	CMpipeIn
LpipeOut	CMpipeOut

6. After being created, the new FPC, with the 2 new pipes FPCpipeIn and FPCpipeOut, sends an ack message to the adapter server, that creates the new VCI and updates its tables in this way:

AdapterServer:

Adapter address	null
Device address	address1a
Bvci	56
Remapped Bvci	null
Vci	1
UpipeIn	FPCpipeIn
UpipeOut	FPCpipeOut
LpipeIn	CMpipeIn
LpipeOut	CMpipeOut

7. The ack message is sent to the lower level through the channel manager:

Ack flag + BVCI + VCI

0 + 56 + 1

8. Gateway 2 handles the ack message and updates its routing tables:

Gateway2:

Adapter address	2.adapterAddress
Device address	address1a
Bvci	34
Remapped Bvci	56
Vci	1
UpipeIn	2.pipein
UpipeOut	2.pipeout
LpipeIn	1.pipeIn
LpipeOut	1.pipeOut

9. It creates a new ack message and sends it to the lower level, swapping the arrived bvci with the bvci in its table:

$$\text{Ack flag} + \text{BVCI} + \text{VCI}$$

$$0 + 34 + 1$$

10. Gateway 1 handles the ack message and updates its routing tables:

Gateway1:

Adapter address	2.adapterAddress
Device address	address1a
Bvci	BVCI_1a
Remapped Bvci	34
Vci	1
UpipeIn	2.pipein
UpipeOut	2.pipeout
LpipeIn	1.pipeIn
LpipeOut	1.pipeOut

11. It creates a new ack message and sends it to the lower level, swapping the arrived bvci with the bvci in its table:

Ack flag + BVCI + VCI

0 + BVCI_1a + 1

The same procedure happens when device 2 sends the binding message.

Chapter 6

Conclusions

We tried to stay as close to the original goals as possible. The software we developed has been unit tested and globally tested in several examples of configurations and situations. Even though the lack of a complete system, from the working physical device to the application level (query and data management) reduced the availability of reliable testing scenarios, we could prove some features of the components we developed.

The whole system is structured in logical levels that abstract a part of the entire behavior. Communication travels through the stack of levels. This allows future modifications to be restricted to a single level.

The system is able to support every kind of device, since it only transmits messages and let the device “tell” how it is going to behave.

Still remain some components to implement in detail and open problems that require careful design. We already described the guidelines for the last components of the system in Sections 3.2 and 4.2. Anyway there are open problems that interest the whole system. One is the possibility of a network failure that causes a channel to stop working. Even though the socket channel can handle a failure, every possible scenario should be taken into account.

Bibliography

- [1] <http://perla.dei.polimi.it>.
- [2] Fortunato Marco, Marelli Marco, *Design of a declarative language for pervasive systems*. Master's thesis, Politecnico di Milano, Milano, 2007.
- [3] Schreiber F.A., Camplani R., Fortunato M., Marelli M., *PERLA: A Declarative Language and Middleware for Pervasive Systems*. Adjunct Proceedings - Posters and Demo Abstracts, 3rd EuroSSC Zurich, 2008, pp. 19-20
- [4] Schreiber F.A., Camplani R., Fortunato M., Marelli M., Pacifici F., *PERLA: a Data Language for Pervasive Systems*. Proc. of the Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2008) Hong Kong, 2008, pp. 282-287
- [5] Schreiber F.A., Camplani R., Fortunato M., Marelli M., *PERLA - PERvasive LAnguage*. Executive Summary. Dipartimento di Elettronica e Informazione - Politecnico di Milano (http://perla.dei.polimi.it/files/documentation/perla_middleware.pdf)