Institute of Parallel and Distributed Systems
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2214

Development of a Framework for Mobile Data Stream Processing

Antonio Fernández Zaragoza

Course of Study:	Infotech M. Sc.
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dipl. Inf. Harald Weinschrott Dipl. Inf. Nazario Cipriani
Commenced:	April 15, 2009
Completed:	October 15, 2009
CR-Classification:	C.2.4, H.3.4, H.2.4, C.1.3, D.1.5, D.2.2

Abstract

Given the growing proliferation of embedded sensors that measure their environment getting continous volumes of data, there is an increasing interest on the possibilities that this continuous streams of information are able to offer.

This kind of data is characterized as a potentially infinite flow of data elements from one or more data sources data stream. As part of this new stream scenario is developed NexusDS, a project that provides a flexible and extensible platform for data stream processing, allowing custom operators, supporting heterogeneous system topologies and managing a flexible data model to support variety kind of data.

In the last few years, there has been an increasing number of mobile devices such as cell phones, PDA and handheld. These devices incorporate sensor technology that can measure environment getting continuous volumes of that, offering continuous streams of information.This thesis focuses on creating a data processing framework for mobile devices and join it to current NexusDS platform, allowing distributed continuous queries over data streams in mobile devices. This new framework incorporation will permit the participation of mobile devices such phones or PDAs in the NexusDS project.

Keywords: ***NexusDS, JXTA, JXME, J2ME, p2p, framework, stream data, data processing***

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Given the growing proliferation of embedded sensors that measure their environment to get continous volumes of data, there is an increasing interest on the possibilities that this continuous streams of information are able to offer.

In the last few years, there has been an increasing number of mobile devices such as cell phones, PDA and handheld. These devices incorporate sensor technology that can measure the environment getting continuous volumes of it, offering continuous streams of information. With different kinds of wireless communication systems, It is possible to feed the continuous flow of information produced by this apparatus into a more powerful processing systems.

The concept of context-aware computing applications is emerging. These applications rely on models of the physical world interacting with them, such as their current location of an user, their proximate physical environment or their activity. This comes from the idea that, at a given moment, the most interesting part of the world around a person is its surroundings, where he or she can interact.

In the last few years, many studies have been focused on techniques for efficient and distributed processing of huge, unbound data streams. This kind of data is characterized as a potentially infinite flow of data elements from one or more data sources data stream. As part of this new stream scenario, NexusDS is developed. NexusDS is a project that provides a flexible and extensible platform for data stream processing, allowing custom operators, supporting heterogeneous system topologies and managing a flexible data model to support variety kind of data.

1.1 Objectives

This thesis focuses on creating a data processing framework for mobile devices and join it to current NexusDS platform together with the framework designed in [10], allowing distributed continuous queries over data streams both on computers and mobile devices. This new framework incorporation will permit the participation of mobile devices such as phones or PDAs in the NexusDS project.

These devices will have different roles within the platform. One case can be in a client with a mobile device that needs to outsource those parts of the logic application that is not capable of processing, delegating this process to more powerful computers. In other situations, a phone can be providing continuous data from a sensor, like its camera. And finally, can participate doing stream processing tasks with other mobile devices or computers, or can host distributed services for the platform. Given this high participation in the platform, we can say that this framework is necessary for enabling the incorporation of mobile devices providing a large amount of functionality in order to achieves the objectives of NexusDS. In this way, this thesis will expose the developing process of this system, with the steps of analysis, design, and implementation of a functional prototype.

1.2 Thesis structure

This chapter has briefly introduces the motivation and objective of this work. Also gives an overview of the NexusDS project as the foundation of the provided framework.

Chapter 2 gives an overview of the different techonologies that will be used during the developing process.

Chapter 3 defines the requirements of our system.
Chapter 4 explains the design of the intended framework.
Chapter 5 describes the prototyping implementation at different levels of detail.
Chapter 6 defines a evaluation test for the framework and gives some results.
Chapter 7 gives a conclusion and an outlook for future research.

1.3 Foundation

This chapter gives an overview of the NexusDS platform as described in [7].

1.3.1 The NexusDS project

NexusDS is a evolution of Nexus system, an open platform for context-aware applications originally designed for the query-response paradigm. NexusDS uses the powerful Augmented World Model (AWM), an extensible data model based on object oriented concepts. With this model is achieved the requisite of having a data model to support variety kind of data: a building, a point of interest or an image... . NexusDS provides a extensible domains-specific operators and services that may even utilize specific hardware available on dedicated computing nodes. A major purpose of NexusDS is to flexibility and seamlessly integrate operators and services allowing for extensions depending on domain and application needs. Operators are push-based and are used to integrate custom data-processing into the system. In contrast, services follow a request-response paradigm and are used to implement customized system behavior in terms of service functionality offering.

1.3.2 NexusDS architecture

As we can see in Figure 1.1 The architecture of NexusDS consists of four layers. The architecture combines a highly adaptively service platform and an extensible operator execution framework.

Communication and Monitoring Layer

This layer inherits the flexibility and scalability from P2P networks There are two basic services:

- The monitoring Service is the component that monitors the system and keeps track of the computing nodes and their specific characteristics.
- The Service publisher Service permits service deployment.

Multiple instances of both modules are created and distributed across multiple nodes to increase availability and to reduce possible bottlenecks.

Nexus Core Layer

This layer defines the core services of NexusDS. The services are distributed among computing nodes dedicated to run the services to avoid a single point failure. The core services are:

- Core Query Service: The Core Query Service accepts query graph as argument and takes the necessary actions for query deployment and execution.
- Operator Repository Service and Core Operators: This component defines the basic operators available in NexusDS. These operators are stored in an operator repository together metadata needed to identify operator-specific constraints to find appropriate computing nodes.
- Operator Execution Service: This service xecutes the assigned query fragments. It interacts with other core services, such as the Operator Repository Service, to fetch missing operators, or the Core Query Service, to signal an overload situation and, e.g., initiate query reoptimization.

Nexus Domain Extensions Layer

This layer builds on top of the Nexus Core Layer and logically clusters services and operators of a certain domain of interest. The provided functionality is tailored to the particular domain.

Nexus Applications and Extensions Layer

This layer enables services or operators that are specific to a single application. It is possible to outsource portions of the ap plication logic to NexusDS.

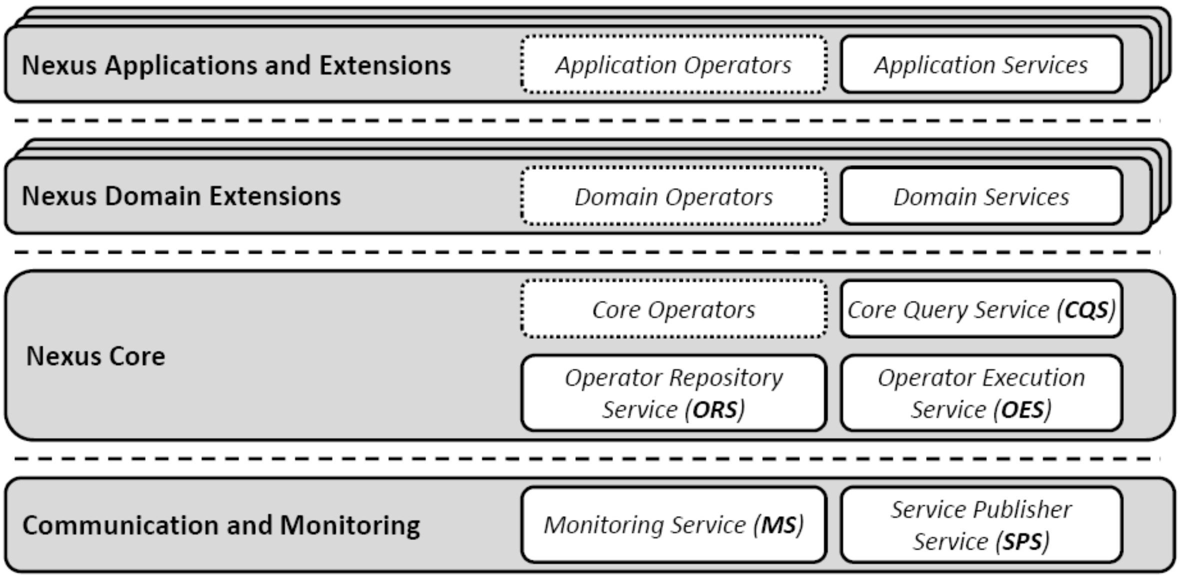


Figure 1.1: NexusDS layers [7]

CHAPTER 2

TECHNOLOGIES

Several software technologies provide us all the necessary support for the develop process of the system here described. The most important of them are explained in this chapter, like J2ME that will be our programming language and platform for mobile devices; Apache Ant and Antenna, two tools to providing a easy way to the software build; and finally JXTA, the project that will be used like connectivity platform.

2.1 Peer to peer network

A peer-to-peer distributed network architecture is composed of participants that make a portion of their resources (such as processing power, disk storage or network bandwidth) directly available to other network participants. As we can see in 2.1 there is not a central coordination instances (such as servers or stable hosts). Peers are both suppliers and consumers of resources, in contrast to the traditional client-server model where only servers supply, and clients consume. There is also a kind of p2p network called hybrid, where one or more peers have special purposes.

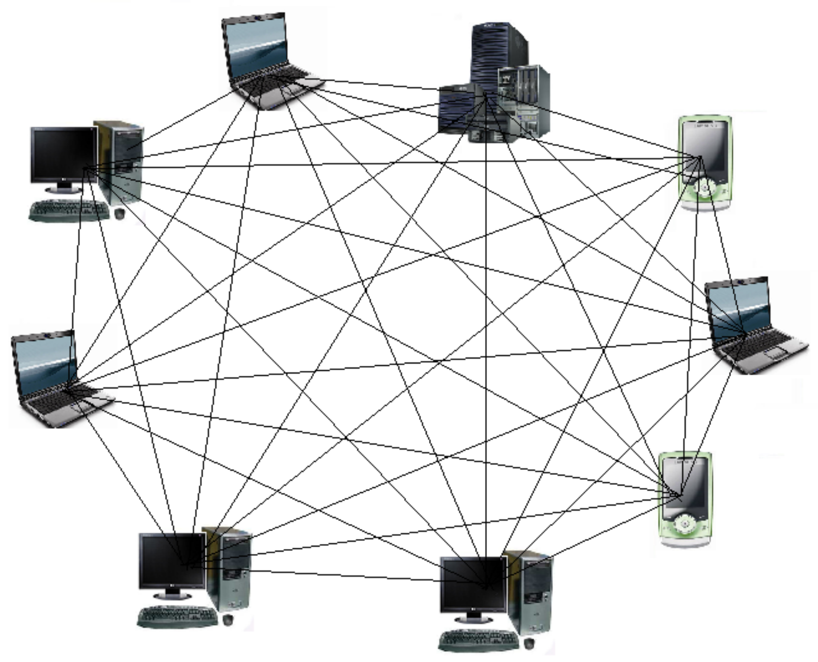


Figure 2.1: Pure p2p network with different devices connected

2.2 Java Micro Edition (J2ME)

For the implementation of this framework, the Java Micro Edition is used. This decision is taken because of the capacities that J2ME offers, like OS independent implementation, mature platform, good supporting for wide range of mobile devices and much more properties and characteristics that are explained in the next chapters. On the other hand, this framework will be integrated in the NEXUS project, build with Java in its standard version. According to the mobile devices with which we work, we select the combination of the CLDC configuration with the MIDP profile. The optional profile jsr75 is also included for efficient accessing to the file system of the devices.

In the next points, an overview and some specifications and concepts are given in order to understand what this version of Java is like and its capacities in the world of mobile devices.

2.2.1 J2ME overview

Java Platform, Micro Edition (Java ME) [2] provides a robust, flexible environment for applications running on mobile and other embedded devices-mobile phones, personal digital assistants (PDAs), TV set-top boxes, and printers. Java ME includes flexible user interfaces, robust security, built-in network protocols, and support for networked and offline applications that can be downloaded dynamically. Applications based on Java ME are portable across many devices, yet leverage each device's native capabilities.

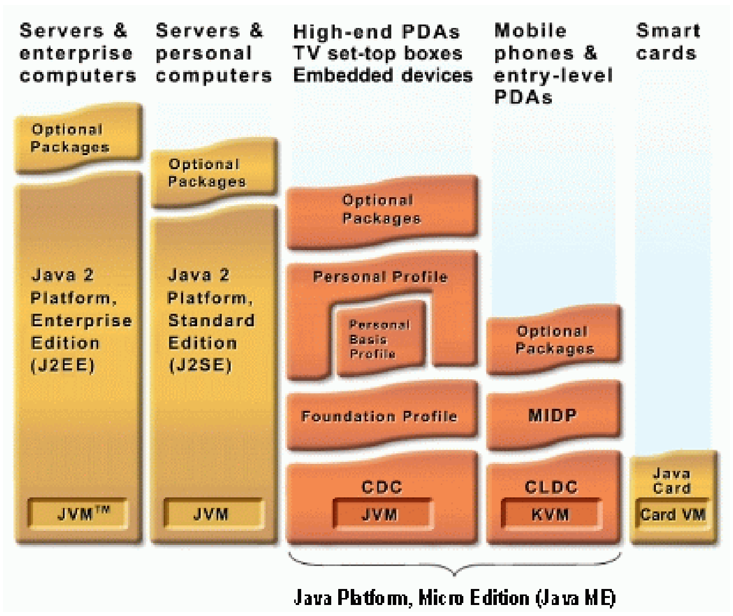

Servers & enterprise computers
Servers & personal computers
High-end PDAs TV set-top boxes Embedded devices
Mobile phones & entry-level PDAs
Smart cards
Optional Packages
Optional Packages
Optional Packages
Optional Packages
Java 2 Platform, Enterprise Edition (J2EE)
Java 2 Platform, Standard Edition (J2SE)
Personal Profile
Personal Basis Profile
Foundation Profile
MIDP
CDC
CLDC
Java Card
JVM™
JVM
JVM
KVM
Card VM
Java Platform, Micro Edition (Java ME)

Figure 2.2: Overview of the different Java Platforms [2]

The figure 2.2 shows the different Java technologies and the principal points where are different between them On the top of the image we can see the platforms where each one are belong to. For example, J2SE and J2EE works with powerful machines like servers, but J2ME only works with mobiles devices with limited resources. Other difference between the Java editions are the Virtual Machine. Due to limitations of the phones and PDAs, J2ME with CLDC configuration has a different virtual machine called K Virtual Machine (KVM) that is adapted for these conditions. Finally, each version has its own API sharing certain group the classes with the others. Referring again to the JXME with CLDC configuration and MIDP profile, that has a small subset of classes of the J2SE version.

Java ME platform is a collection of technologies and specifications that can be combined to construct a complete Java runtime environment specifically to fit the requirements of a particular device or market.

The Java ME technology is based on three elements:

- A configuration provides the most basic set of libraries and virtual machine capabilities for a broad range of devices.

- A profile is a set of APIs that support a narrower range of devices.
- An optional package is a set of technology-specific APIs.

2.2.2 Connected Limited Device Configuration (CLDC)

This configuration is to fit small mobile devices. The figure below represents an overview of the components of Java ME technology and how it relates to the other Java Technologies.

It is specifically designed to meet the needs of a Java platform to run on devices with limited memory, processing power and graphical capabilities. On top of the different configurations Java ME platform also specifies a number of profiles defining a set of higher-level APIs that further define the application. A widely adopted example is to combine the CLDC with the Mobile Information Device Profile (MIDP) to provide a complete Java application environment for mobile phones and other devices with similar capabilities.

For a CLDC and MIDP environment, which is typically what most mobile devices today are implemented with, a MIDlet is created.

2.2.3 Mobile Information Device Profile (MIDP)

Combined with the Connected Limited Device Configuration (CLDC), MIDP provides a standard Java runtime environment for today's most popular mobile devices, such as cell phones and personal digital assistants (PDAs). MIDP is widely adopted as the platform of choice for mobile applications.

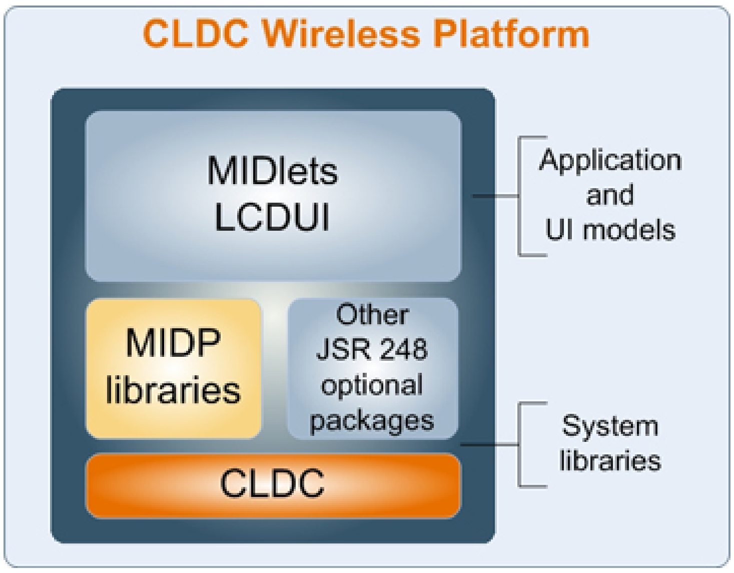


Figure 2.3: Common combination of the CLDC configuration with MIDP profile [2]

2.2.4 JSR75 profile

This is a optional profile that specifies two packages, the PIM package and the File Connection package. The last one is the package (javax.microedition.io.file) that will be included together with the configuration CLDC and the profile MIDP.

The File Connection package provides classes to accessing the mobile device file system, even removable storage devices like memory cards. In this way, enables the accessing to data such images, audio files, videos and more.

The inclusion of this profile is thought in order to give flexibility to manage different kind of files, starting with configuration files.

2.2.5 MIDlet application

A MIDlet is the application created with Java Micro Editon. A MIDlet can be written once and distributed in order to run in every device conforming with the specifications according with the configuration and profiles selected, as well as general specifications of J2ME. The MIDlet can reside on a repository somewhere in the ecosystem and the end user can search for a specific type of application and having it downloaded over the air to his/her device. The distribution of MIDlet consist of:

MIDlet jar The .jar file contains all the packages and classes of the application and other resources like images, sounds... etc. At least one of the classes must extend the class javax.microedition.midlet.MIDlet and can be called the main class of the MIDlet. This file is the main file of a MIDlet application distribution and obviously, is mandatory.

MIDlet jad The .jad file is a description file that contains information about the MIDlet and the file .jar. Some information can be the location and size of the .jar file or the version of MIDP and CLDC. This file is no mandatory but is recommendable and in some devices is necessary for a correct installation for the MIDlet.

In the figure 2.4 we can see that building the MIDlet is a bit different of a classical Java program build and include several steps that are:

1. Compilation of the java classes with the java compilator.
2. Package of the MIDlet application in the .jar file and creation of the .jad file.
3. Preverification of the file .jar. This preverification can be done over the classes before the packaging. This preverfication ensures that there is not malicius code that can damage or present a strange behavior in the device or virtual machine.
4. Deploy the MIDlet in the device, specifying directly the .jar (from file system or download over the air) file or indicating the .jad that contains the information about the .jar file.

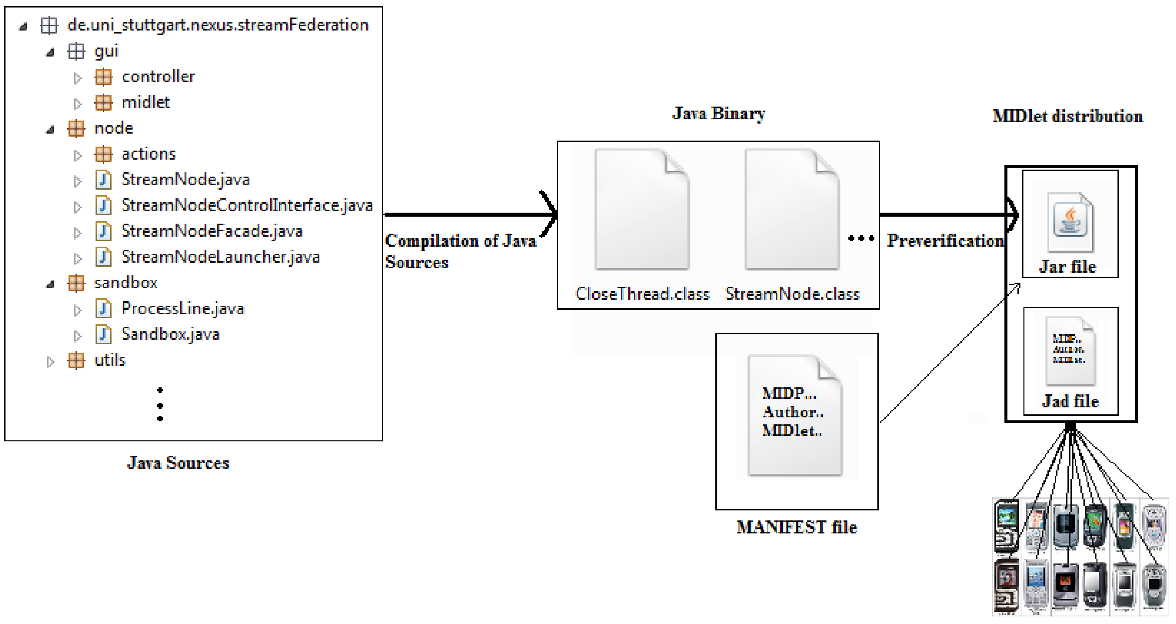


Figure 2.4: Build process of a MIDlet distribution

2.2.6 Target devices

CLDC [2] is designed to bring the many advantages of the Java platform to network-connected devices that have limited processing power, memory, and graphical capability, such as cellular phones, pagers, low-end personal organizers, and machine-to-machine equipment. In addition, CLDC can also be deployed in home appliances, TV set-top boxes, and point-of-sale terminals. Target devices typically have the following capabilities:

- Processor: 16-bit or 32-bit with a clock speed of 16MHz or higher.
- Non-volatile memory: At least 160 KB allocated for the CLDC libraries and virtual machine.
- Memory: At least 192 KB of total memory available for the Java platform.
- Power consumption: Low, often operating on battery power.
- Connectivity: Some kind of network, often with a wireless, intermittent connection and limited bandwidth.

2.2.7 The K Virtual Machine (KVM)

The K Virtual Machine (KVM) [16], a key feature of the J2ME architecture, is a highly portable Java virtual machine designed from the ground up for small- memory, limited-resource, network-connected devices such as cellular phones, pagers, and personal organizers. The KVM can be deployed flexibly in a wide variety of devices appropriate for various industries and the large range of trade-offs among processor power, memory size, device characteristics, and application functionality they engender.

2.2.8 Limitations introduced by J2ME

Before the design phase, we can see some limited or restricted points due to Java Micro Edition, In short, the classes of J2ME contains only part of J2SE API; there are no parametrized values, generics and it doesn't support reflection. The exception-handling capacity is limited and creating user throwable exceptions is not allowed Finally, the capacity of the virtual machine KVM is adapted to the conditions of the mobile devices, with limited memory and resources.

Along the thesis, some of these restrictions will be explained for concrete cases in the design phase and implementation phase.

2.3 Apache Ant

Apache Ant is a software tool, platform-independent, for automating software build processes. It is similar to Make but is implemented using the Java language, requires the Java platform, and is best suited to building Java projects. Ant uses XML to describe the build process and its dependencies, offering a large number of built-in tasks that can be used without any customization. This tool is used in order to make easy the build process of the MIDlets with the task provides by Antenna that is explained in the next point.

2.4 Antenna

Antenna [14] provides a set of Ant tasks suitable for developing wireless Java applications targeted at the Mobile Information Device Profile (MIDP). With Antenna, you can compile, pre-verify, package, obfuscate, and run your MIDP applications (aka MIDlets) and manipulate Java Application. These tasks can be done with the J2ME Wireless Toolkit or other Java IDEs, but using an Ant script results in a defined and reproducible build process that is independent of a particular programming environment.

2.5 JXTA

In [10] is discussed why the JXTA platform is used like connectivity platform and some of the advantages that were taking in account are:

- JXTA provides a P2P generic structure.

- Direct connection between available nodes in the P2P network.
- Firewall systems, proxys and security services.
- OS, Platform, and Application independent along with suport for PDAs and Handheld devices.

We can see that the last point refers to PDAs and Handheld devices, just the mobile devices where our framework is desinated to. This decision taken in [10] make possible integrate our system easily, re-use design and also code. There is not motive to change this platform for other.

In the next points will be explained the JXTA project in general and its JXME binding, as well as the two versions for JXME: proxied and proxyless version. At the end of the chapter is exposed the version selected and the motivations.

2.5.1 The JXTA project

Project JXTA [1] is an open network computing platform that defines a set of generalized P2P protocols which enable any type of networked device like sensors, cell phones, PDAs, laptops, workstations, servers and super computers to cooperate and work together as peers. The advantage of the JXTA protocols consists in the fact that they do not depend on programming languages as they are implemented for different environments. Such implementations are also called bindings. The most important consequence of the common use of the JXTA protocols is their total interoperability.

The JXTA goals and requirements

The main goals of Project JXTA according to [1] are:

Interoperability JXTA technology aims to allow peers find other peers and interconnect with them without depending on network addressing and physical protocols.

Platform independence JXTA technology is designed to be independent of programming languages such as C or Java, system platforms and networking platforms such as TCP-IP or Bluetooth.

Ubiquity JXTA technology is intended to be accessible by any digital device such as sensors, consumer electronics, PDAs, appliances, network routers, desktop computers, data center servers and storage systems.

[1] literally states that any application using the JXTA technology must be able to:

- Find other peers on the network with dynamic discovery across firewalls and NATs.
- Easily share resources with anyone across the network.
- Create a group of peers that dynamically cooperate to provide a service.
- Monitor peer activities remotely.
- Securely communicate with other peers on the network.

The JXTA concepts and elements

Hereafter will be described the primary components of JXTA.

Peers

A Peer in JXTA is any entity which implements one or more of the defined JXTA protocols. A PC, a server, a smartphone or a simple sensor array with network capabilities can be a JXTA Peer. Each one of them must operate independently and be uniquely identified by a name: the Peer ID.

Peers advertise at least one network address where they can be reached. This address is published as a Peer Endpoint. Peers can directly connect to other Peers by using these endpoints. Sometimes a point-to-point communication is not possible (e.g. different physical layers) arising the need of intermediaries. The use of intermediary peers does not affect a JXTA application.

Depending on the role on the network, and the JXTA Services implemented, we can distinguish different kind of peers:

Minimal-Edge Peers are those peers which implement only the core JXTA Services, 5 relying on other peers to fully participate in a JXTA network where more services are available. This is usually the case of sensor devices.

Full-Edge Peers are those peers which implement the core and standard services of JXTA, being able to participate in all protocols. Most of the peers in a network are of this type (PCs, phones, servers...).

Super-Peers are those peers which provide Services and resources to support a JXTA network. The specification defines three key functions for Super-peers.

- Relay is a superpeer that stores and forwards messages from peers which have no direct connectivity to the final destination.
- Rendezvous is a superpeer which handles broadcast of messages, and indexes the advertisements, assisting the Minimal-Edge peers to find proxies to provide further functionality.
- Proxy as already stated, translates and summarizes in both ways the requests from minimal-edge peers.

Peer Groups

Although it could be possible to describe all the network connected systems as a huge and unique P2P system, it is more accurate and practical to describe it as a set of partitions, or group of peers, which can overlap. In JXTA a Peer Group is a collection of network peers which share resources and services. No instruction or recommendation is given in the JXTA specification about how this partitioning must be done.

A Peer Group in JXTA is designed to be as generic and unconstrained as possible, allowing peers to easily join any group and to be members of as many groups as desired. Again, the JXTA specification does not state anything on how membership should be kept or maintained, leaving the form under which it is implemented open.

Peer Groups help to create an environment which can be more easily controlled than a larger network, which is useful for organization, monitoring or supervision in large scale scenarios.

Network Services

Usually Peers join a Peer Group to use the Services available within it. The most simple of these Services is the Membership Service. There are two types of JXTA Services:

Peer Services are services available only on the peer that is advertising that service. If the peer is down (or leaves the group) the service is no longer available to the group. Several peers can provide different instances of a service, but they are different services, which are independently published.

Peer Group Services are services provided by a collection of peers, cooperating among them. If one of the peers fails, the service remains unaffected as long as there is still one peer providing the service. Note that not all peers of the group must provide a Peer Group Service.

JXTA defines a core set of peer group services which must be implemented by every peer in order to join a Peer Group. Note that peers can interact via service only if they are members of the same group. Those main or core services are:

Endpoint service is the service used to send and receive messages.

Resolver service is used to send generic requests to other peers, such as find- ing advertisements.

Additionally to the core services, JXTA provides another set of services called standard services that are built on top of the core services. They are found in most Peer Groups:

Discovery service used to find Peers Peer Group Services, Pipes an others resources.

Membership service used to set up secure identities in a Peer Group helping to restrict peer accesses to Services, Peer Groups, etc.

Access service used to validate requests from one peer to another. Not all actions require this step, only those limited to some Peers whose credentials need to be checked.

Pipe service used to create and manage the Pipe connections between members of a Peer Group.

Monitoring service used to allow supervising operations of other members of the Peer Group.

This Services hierarchy can be seen in Figure 2.5

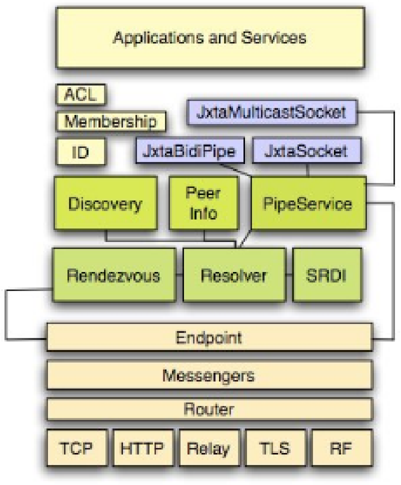


Figure 2.5: Services hierarchy on the JXTA platform [10]

Modules

JXTA modules are a low-level JXTA abstraction used to represent any piece of "code" and the interface (API) which that code provides. Modules are used to implement services, message transports and other loadable bits of JXTA code, providing a generic abstraction to allow a peer to instantiate a function or service. The module abstraction includes a module class, module specification, and module implementation.

When a peer joins a peer group they may find new behaviors that they may want to instantiate. For example, when joining a peer group, a peer may be required to provide a new search service that is only used in this peer group. In order to join this group, the peer must instantiate this new search service. The modules allows the description and publication of platform-independent behavior, and is essential to supporting the development of new peer group services which are provisioned by a heterogeneous cadre of peers.

Most JXTA developers do not typically have to deal with modules as distribution the includes the initial set of services required by most applications.

Pipes

Pipes are the way defined by JXTA to communicate information between Peers. Pipes are unidirectional and asynchronous and do not require to have a Peer at the end, and they can change along the time (i.e. redundancy of communication or services). Pipes are only possible within the scope of a Peer Group.

A Pipe opened by one peer can direct to another peer (point to point), or to more than one (propagate pipe). The destination of the pipe is called the End Point. Pipes are not bound to a physical network address, while End Points are. If several pipes terminate at the same End Point, Messages are queued on the destination peer.

Pipes are very-low level JXTA communication mechanisms, and it is recommended to use bidirectional high level abstractions such as the JxtaBiDiPipe and JxtaSocket when possible.

Messages

Messages are the data encapsulation to be sent through Pipes from one Peer to another. JXTA defines each message to have an envelope and a body. The envelope contains a header, source and destination End Points information in URI form, and optionally a message digest. The body can contain an arbitrary amount of information. If credentials should be sent, they must be included in the body. In order to provide an easy-to-parse encoding mechanism, messages are written in XML format. The specification does, however, not close the door to the use of non-XML data in the message. In fact, the JXSE reference implementation is able to make use of a binary format defined in [15] to send the messages through the Pipes.

Advertisements

All JXTA resources are represented as Advertisements that are also encoded as XML documents. The working mechanism for sharing and discovering Advertisements is very similar to the DNS service on the Internet. Each Advertisement is published with a lifetime, which enables to use decentralized directories. JXTA defines different Advertisement types for Peers, Peer Groups, Services, Pipes, etc. and allows programmers to create their own customized ones.

The JXTA protocols

The core JXTA P2P interaction model is completely expressed as a set of simple protocols specifying to the last detail all the connectivity among peers. The specification of these protocols is also open, allowing any interested party to implement them with any programming language for any platform. The JXTA specification does not require that all peers implement all of these core protocols but only those that it actually uses.

The JXTA core protocols are:

Peer Discovery Protocol (PDP) Used to discover advertisements from other peers within the peer group; useful for discovering peers, peer groups, pipes, and services.

Peer Information Protocol (PIP) Used by a peer to obtain status information about another peer.

Peer Resolver Protocol (PRP) Used by a peer to send generic queries to one or more peers, and receive responses; this protocol is the base of the PDP and the PIP.

Pipe Binding Protocol (PBP) Used by peers in binding themselves to a pipe endpoint, needed to open a Pipe to another peer or set of peers.

Endpoint Routing Protocol (ERP) Used to provide routing information for paths between peers when a direct connection is not possible.

Rendezvous Protocol (RVP) Used by edge peers to resolve, propagate and advertise resources, and by rendezvous peers to organize themselves, share the distributed hash table address space and propagate messages.

2.5.2 Jxta-Jxme

The JXTA Java Micro Edition [12] provides a JXTA compatible platform on resource constrained devices using the Connected Limited Device Configuration (CLDC) or the Mobile Information Device Profile 2.0 (MIDP), or Connected Device Configuration (CDC) . The range of devices include from smart phones to PDAs. Using JXTA Java Micro Edition platform, any CLDC/MIDP/CDC device can participate in the JXTA network with any other JXTA device. Currently, JXME has two versions: proxied and proxyless version.

Proxied version

The proxied version requires a Proxy for the devices to be able to participate in JXTA networks. It is necessary to configure a JXTA peer to act as Proxy and here is a full dependency on the Proxy, so the proxied version is not completely peer to peer. Figure 2.6 shows how the JXME peers interact with the JXTA network through JXTA proxy peer, that could be a single point of failure or a bottle neck. For example, the group C of mobile devices lose the connection because their Relay Peer Machine is disconnected.

Even if there were backup Proxies, the end-user would have to manually enter the IP address of the backup servers and restart the entire application.

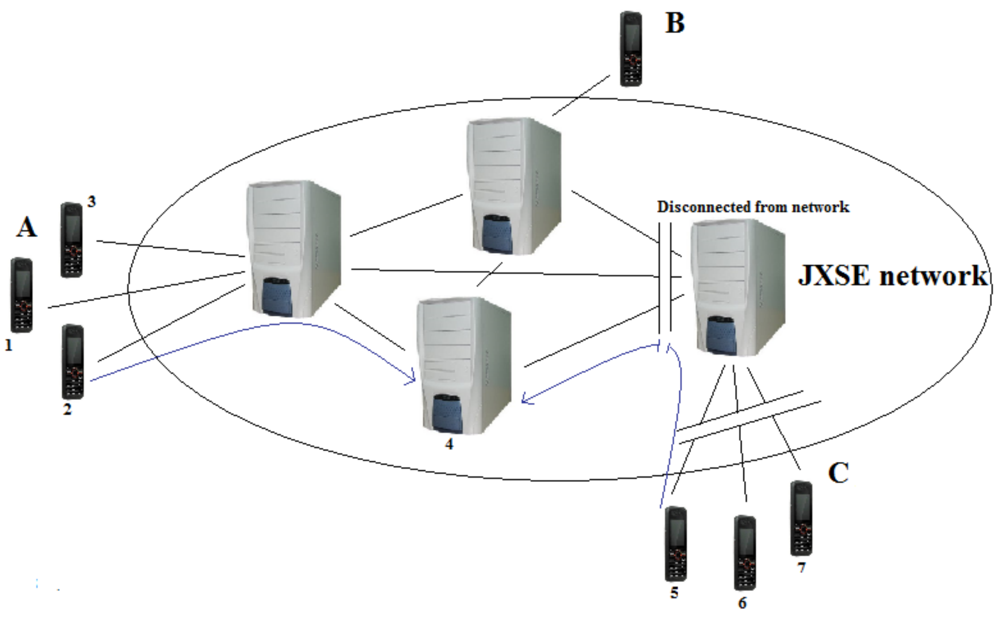


Figure 2.6: JXTA network with both JXME and JXSE peers

From a programming point of view, JXME is simply an API. It has only three classes: PeerNetwork, Message and Element. PeerNetwork is the main class, an instance of this class is a JXME peer. A Message is simply the class that represents messages sent to other peers and is made up of one or more Element instances. The only method that actually accesses network resources is PeerNetwork.Poll() except for PeerNetwork.Connect() which connects to the Proxy.

Advantages for using the proxied version:

- It is a mature version (compared with the proxyless version).
- This version is well documented and has real examples.
- The learning process is not too complex.

Drawbacks for using the proxyless version:

- Very limited functionality and simple API.
- Dependence upon the proxy, single point of failure.
- Slowly, the proxy could be a bottle neck.
- Project stalled.

Proxyless version

[12] On the other hand, there is the proxyless version that has full mobile to mobile and mobile to desktop access. A proxy is not required and has compatible APIs with JXTA Standard edition. In the figure 2.7 we can see the stack of JXME proxyless version. On the bottom of the JXME stack is the MIDP profile. Over this profile is implemented GZIP capacity and some collections that are no available in MIDP profile or CLDC configuration. The next level shows the core protocols described before: peer discovery protocol, peer resolver protocol, pipe binding protocol, rendezvous protocol and on the bottom of all, the endpoint protocol. Finally, on the top are the high level abstractions JxtaSocket and JxtaBiDiPipe that work using the protocols before commented.

In terms of connectivity, it implements UDP pipes like channels, unicast or virtual multicast, and supports JxtaSocket, JxtaBiDiPipe and JxtaMulticast. The methods for addressing consist of a 128Bit UUID peer and group

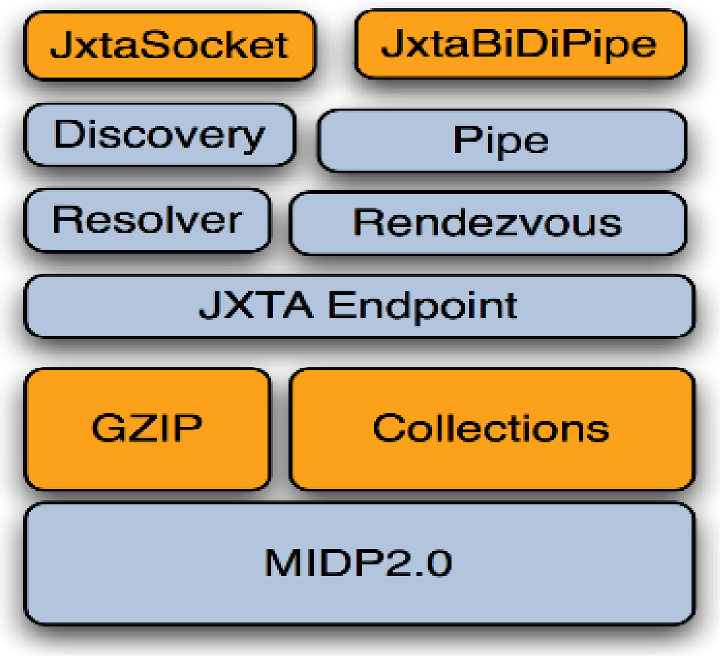


Figure 2.7: JXTA Java Micro Edition Stack [12]

identifier, endpoint addresses, and x509.v3 peer certificate. For security, group membership/ACL and end-to-end TLS transport are available.

Advantages for using the proxyless version:

- Real peer to peer system and directly iteractuation with JXSE peers and other JXME peers.
- Flexibility and autonomy.
- API similar to JXSE.

Drawbacks for using proxyless version:

- Lack of documentation and examples.
- The support is practically nonexistent.
- No mature implementation, not really tested and with several bugs. This version was released one year ago.
- Quite complex architecture.
- All the problems above result in a slow and difficult learning process, spending time fixing some critical bugs that really are not the focus of this thesis.

2.5.3 Choosing version

The decision of which of the two versions should be chosen is not very clear. Both versions have advantages and drawbacks and it is necessary to attend to the most important and real requisites.

One idea is try to carry out a framework with similar concepts, architecture and elements like the current framework for no mobile devices. This will be helpful to integration of the framework in the NexusDS platform, enabling a better compression by all developers. The proxyless version offers an API which is very similar to the JXTA standard, offering the possibility of achieving the previously explained idea.

A second point is the current and future perspective. Nexus is a project with a large future vision, so it makes no sense selecting deprecated projects. Although currently the proxied version is more mature, with more documentation and examples, it is stalled and no longer supported. On the other hand, the support for the proxyless version is poor and with several bugs, some of them essential for correctly working and which must be fixed, but it is the project for JXME that will be continued.

Finally, according to the Architecture of the Federation layer, the proxied version is not a good approach; the proxy is a single point of failure. In fact, the proxied version is not a real peer to peer and offers a poor flexibility for the project. The image 2.8 shows the idea of a transparent peer to peer JXTA network, without differentiating the type of device behind each peer, the connectivity type or the environment.

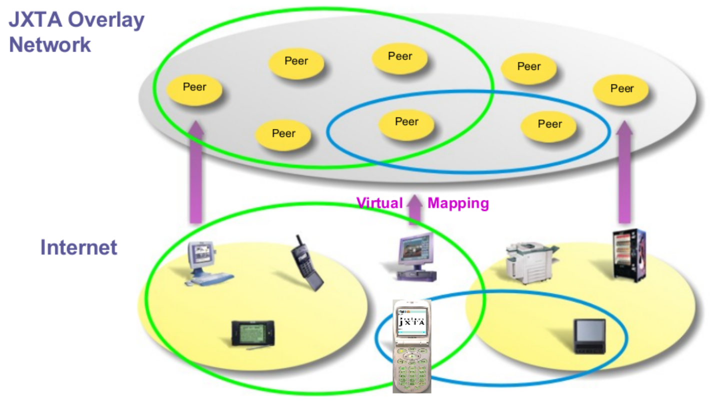


Figure 2.8: JXTA Overlay Network

Taking account the reflexions above explained, is decided to use the proxyless version for the connectivity platform of the system, taking the advantages that offers but also with all its drawbacks associated.

2.5.4 Limitations introduced by JXME proxyless

Before continuing with the design and requirements analysis, the capacities of the technologies selected before must be revised. There are some critical parts of the project where a badly performed previous analysis can be fatal during the design and implementation phases.

As previously was commented, the connectivity network selected was JXME proxyless version, although this project is not mature. For this reason and for the learning process, before starting with the design, some small test applications were implemented in order to see if this version really works correctly. With these tests we are trying to see if the proxyless version could accomplish the goals and capabilities of JXTA. Some objectives of these tests were the discovery capacity, peer and socket connection, creating and joining groups, and even the basic configuration of a peer. Among others, the following errors were discovered:

- Unable to create and join new groups.
- TCP connections problems, enabling only one connection and crashing when trying a second one.
- Socket connection problems due to the impossibility to select a correct messenger.

These problems introduce a new task, that originally was out of the scope of the thesis, consisting in fixing the problems allowing the basic functionality for the thesis.

CHAPTER 3

REQUIREMENTS

3.1 Introduction

The system must be capable of managing information that comes from a stream channel, for example, a sensor. A data stream cannot be saved in a database because of its possibly infinite nature. These streams must be processed on the fly, in real time, creating an output stream that is sent to the client or another node. The client processes and manages the flux of information in different ways depending on the nature of the application or program: it can checked for important events and display information on an screen. In short, the approach is query-stream.

The framework is generic enough to host a wide range of operations, differentiating this system from others where their use cases are limited, with constraint operations and a few specific purposes, and normally without stream processing. On the other hand, the mobile platform where our system will be running introduces some limitations over the operations that will be able to execute. Therefore, the requirements must be analyzed in order to achieve the objectives of the framework being coherent with the restrictions that we have due to mobiles devices.

3.2 Use cases

According to the concept of context-aware application, the situation of the real-world will be taken into account. Therefore, there is a heterogeneous set of scenarios and the information related with them, such as the geographic situation of a user, the activity in his environment, etc. The next situations describe some examples of use cases where the framework will participat together others components of the platform NexusDS, like the framework described in [10].

- A user wants to see on his phone a map containing the changing position of his and those of his friends who are within 500 meters of his location. Here our system can be running in the phone of the user with a role of a client, and if the friends are using phones with GPS device, our framework can also work there, sending and managing the geographic situation like stream information. Perhaps some operations can be done over these data, for example, composing the map tiles with the situation of only the people that acomplish with the 500 meter restriction. This operation can be complex, therefore, probably is executed in more powerful computers and using the framework of [10]. Finally, the client will receive a stream of maps including the situation of his friends. In the figure 3.1 is shown how can be the scenario and which elements participate.
- Enhance the GPS navigation for a car, taxi or bus driver with the real time traffic and traffic predictions according to the weather forecast, by calculating the best route to a destination. In this case, our system can be working in the GPS navigation device with the role of client and we can think that also like a sensor. The device can solicitate the new service and send stream data with its situation, the service will require the necessary traffic and forecast information and some operations will be made over the data in order to send it to the client like another stream data.
- A nurse wants to monitor on a handheld the health parameters along with the position of the residents of an Assisted Living Facility. Here the framework can be running in the handheld and perhaps the health and position come from sensors.
- In an off-road endurance race like the Dakar race, it is required to have a real time video showing the position and the path of the racers. In this situation, complex data and operations must be done, probably using several computers over the framework described in [10], and then, a stream data with the final information that really needs the client is sent to the client, that can be a mobile device connected and working using our system.

- In a disaster situation, monitoring the state and situation of emergency equipment in order to manage the operation from different points. Most of the communication infrastructure is likely to be damaged, so using mobile devices with some kind of wireless communication might be necessary.

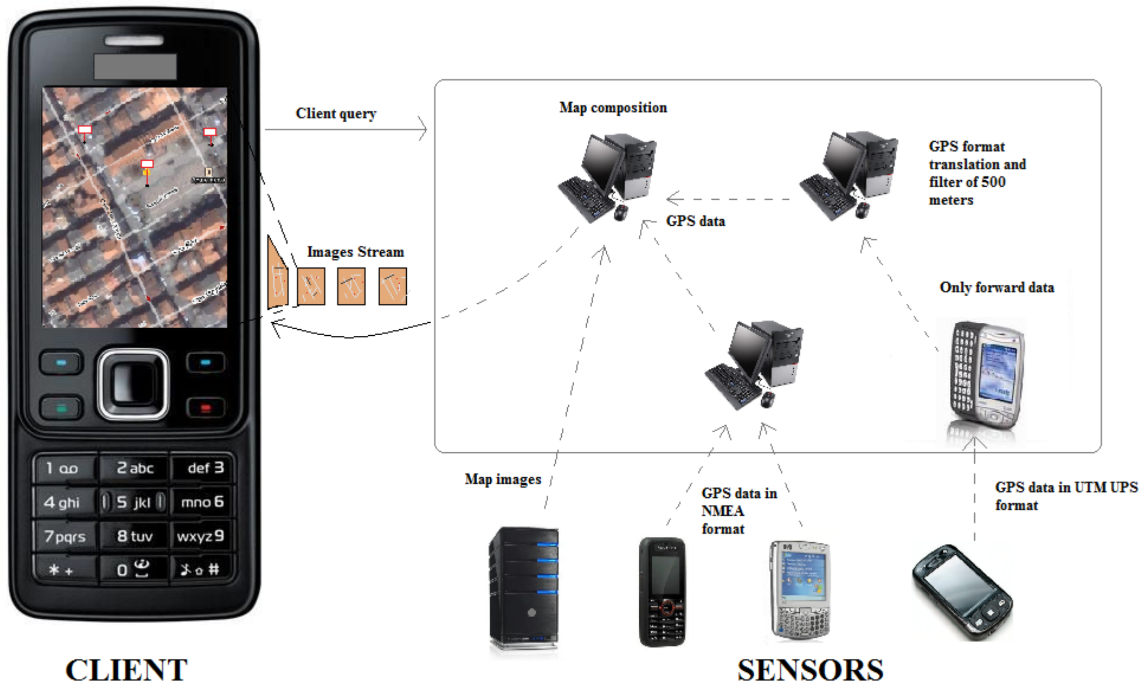


Figure 3.1: Use case where we can see the different roles of a mobile device with the new framework

As we can see, the use cases above work using different kind of data, operations and purposes, but these scenarios share some common properties: the information requested is presented to the user in the form of a data stream which requires the use of operations, some times expensive resource-hungry operations such as image or video rendering. On the other hand, different mobile devices participate in all the scenarios, with a role of client, perhaps transmitting data from their sensors or even participating in computing operations over the stream data and working together with other more powerful computers.

3.3 System requirements

Given the common properties identified above, the next requirements are defined in order to accomplish the objectives of the framework and carry out tasks such as those described in the previous points.

Optimal execution strategy

Among others, there are two strategies for the execution of the different subtasks. The first one is a serial execution, where only one subtask is executed at the same time, and the second one is a parallel execution, where several subtasks can be running at the same time, even if there are some data dependencies between them. Using this last approach, the throughput of the system is increased, but involves computational costs. The framework that is being developed is designed for working in mobile devices, where the CPU and memory is limited.

Therefore, a optimal strategy must be found in order to try to achieve good compute results without saturate the devices. We can provide the system with the capacity of select the appropriate strategy depending of the complexity of the operations or analyzing the capacities of the device.

Set of operations

The system must define a wide range of operations, but only a set of them can be hardcoded at the Stream Nodes due to the unavailable capacity for dynamically downloading new operations and incorporating them to the set. This is because in the platform Java Micro Edition is not possible to download the code and load the operations classes dynamically.

Push-based communication

In order to answer the query of a NexusDS client, the stream of data produced by several sensors must be somehow processed and the product of this transformation is offered to the client as another stream of data. Due to the nature of this data, it makes no sense pulling it continually, therefore the user needs to receive the resulting stream of information as a stable flow. In some cases, the transformation process needed to produce this stream can be quite complex and has to be done in real time. In such situation, it is quite probable that several other streams also need to be processed at the same moment.

Therefore the communication from the sensors to the processing nodes, among the processing nodes, and from the nodes to the clients must be push-based.

Distribution of subtasks in different nodes

In order to avoid possible saturation of nodes or workstations, the stream processing must be divided in subtasks which can be distributed among different nodes. Spreading the work increases the available computing power and resources but, at the same time, introduces some communication and programming overhead. In any event, by distributing the work among several nodes, the normal workstations can act as processing nodes, avoiding the need of very expensive supercomputers. Even the user computer can collaborate with the processing, if it is powerful enough.

Optimization of response times

The time lapse taken between the moment the user issues a request and gets a response from the system, called response time, needs to be sufficiently small so that a user feels that the system is reacting instantaneously.

OS independent implementation

The system must allow the joining of heterogeneous computers. In order to achieve this requirement, an independent design of the operating system is necessary, or if not possible, at least one that can cover the majority of operating systems.

Adaptive system

In the system there will be several nodes and the work must be distributed over these. These nodes must be able to dynamically join and leave the network, with the synchronization and consistency problems inherent. Therefore, special measures must be taken in order to deal with this problem and create a real adaptive system.

Active discovery of networked resources

When distributing the work, it is important to know which nodes form part of the network at a certain time, as well as its available hardware and software resources. For this reason, some kind of discovery mechanism is required.

Scalability

The number of connected nodes is not limited, so system scalability must be guaranteed, thus avoiding problems or degradation of the system with increasing.

Failure tolerance

The system needs to offer a fault tolerance level in order to satisfy the availability requirement of the wide range of tasks and operations that take place in the system.

Attending to the environment where the application works, such as in mobile devices, some basic requirements must be accomplished in order to ensure a stable and reliable system.

Low CPU and memory usage

The developed system has to run with few requirements of CPU and memory as is required by the limitations of mobile devices.

Low bandwidth usage

Also the connectivity properties need to be adjusted to the capacities of the mobile devices and the kind of wireless communications that usually will be used.

Code optimization

The code must be reduced and optimized as much as possible in order improve the performance of the framework running in mobile devices. The size of byte codes must be reduced and that is possible following some points:

- Create the minimum number of objects.
- Re-use objects (pool of objects).
- Avoid a loop if really is no necessary and the operations can be done in other way.
- Reorganization of the loops.
- Reduce parameters of methods.
- Avoid synchronization methods if is possible.
- Avoid complex structures or collections.
- Close file or network connections when are not used in order to free memory an recurses.

Of course there are much methods and most of these strategies are common in code optimization and can be applied in different programming languages and platforms, but in the world of mobile devices, these points are very important and can make the difference between a efficient application and a poor application.

Supporting for wide range of mobile devices

According to the specifications of the mobile devices (CPU, memory. connectivity...), the system must be capable of running in a wide range of mobile devices if the operating system independence is accomplished.

CHAPTER 4

DESIGN PROCESS

The design process of the framework is defined in this chapter. The first point defines the global architecture that follows the system and then the principal components are defined including their objectives inside the framework, their particular architecture and their own elements.

In previous chapters it was explained that the framework here modeled is destined to work integrated in the NexusDS project, therefore, some core components belonging to this platform will be commented in order to see how our framework will interact with the rest of the system in order to understand where the framework is fit in and the role it fulfils.

4.1 NexusDS concepts

Here are described the components and services [7] that will manage the petitions from the clients, receiving the queries and translating them into a graph of operations. We can see an example of how works these services in the Figure 4.1. The most suitable operations can be selected from an operation repository and the graph will be deployed into the most suitable stream nodes of the network. Another task looks for partial query graphs in a query repository in order to reuse them. Most of these components are implemented in a distributed way taking advantage of the P2P protocol that the Stream Platform provides.

4.1.1 Query Service

Query interface

This element receives the client queries. Here a query repository can be consulted in order to try and re-use some previous queries if possible.

Query optimizer

It carries out a first optimization, allowing influence the logical optimization process by plugging in additional optimization strategies.

Query fragmenter

Here the second optimization is made. Once it receives the query, it checks if the query is executable looking for suitable data sources and physical operators using the Operator Repository Service that ask the operator repository. Finally, the suitable nodes for running a certain operator must be selected. After doing these task, it tries to find a optimal execution plan and fragments the query according to the select computing nodes and strategies.

Execution manager

This component distributes the fragments and keeps in synchronization with the Operator Execution Service. This is necessary since in general system conditions vary over time, making it a necessity to re-optimize the execution plan.

4.1.2 Operator repository service

This component defines the basic operators available in NexusDS. These operators are stored in an Operator Repository together with new ones provided by third companies. The Operator Repository needs to store some description of the operation, and also the operation itself or an URL describing where to find it. In this way, it will be able to look for an operation given its characteristics or functionality, or just an identifier. This component will be the easier one to distribute over the Stream Platform by implementing a Network Service and adding it to the services provided by those Stream Nodes that are powerful enough.

4.1.3 Monitoring service

It is the component that monitors the system and keeps track of the computing nodes and their specific characteristics.

Not all the Stream Nodes in the network have the same hardware resources, process power and network connection. Also each one of them will have different CPU and memory load at a given moment. For this reason, checking what Stream Nodes are attached to the network and running will not be enough. Information regarding the nodes status and characteristics must be taken into account.

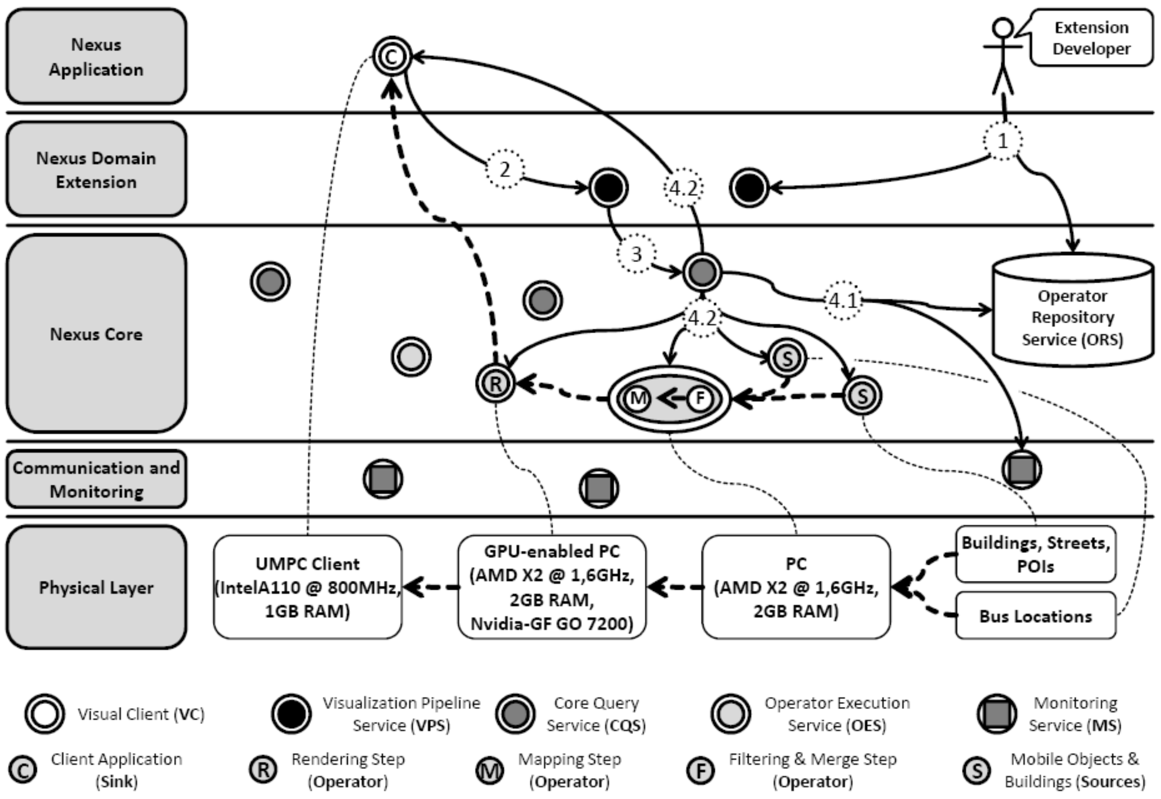

Nexus Application
Nexus Domain Extension
Nexus Core
Communication and Monitoring
Physical Layer
Extension Developer
1
2
3
4.1
4.2
4.2
Operator Repository Service (ORS)
UMPC Client (IntelA110 @ 800MHz, 1GB RAM)
GPU-enabled PC (AMD X2 @ 1,6GHz, 2GB RAM, Nvidia-GF GO 7200)
PC (AMD X2 @ 1,6GHz, 2GB RAM)
Buildings, Streets, POIs
Bus Locations
Visual Client (VC)
Visualization Pipeline Service (VPS)
Core Query Service (CQS)
Operator Execution Service (OES)
Monitoring Service (MS)
Client Application (Sink)
Rendering Step (Operator)
Mapping Step (Operator)
Filtering & Merge Step (Operator)
Mobile Objects & Buildings (Sources)

Figure 4.1: NexusDS Processing Model for the Visualization Application Scenario [7]

Query Service use the Monitoring Service to get most appropriate node, or set of nodes, to host and run an specific subgraph of operations. This functionality could be distributed among several nodes by using a Network Service reducing possible bottlenecks. The service as a whole entity will receive a description of the subgraph of operations to be executed, and will find a subset of the connected Stream Nodes that will be able to carry out this operations and are quite idle in that particular point it time.

4.2 General architecture

The architecture of our framework consists of two entities, the Stream Node Pool and the Stream Platform. Figure 4.2 shows the relation between these main components and some core components of NexusDS with relation to the Stream Node Pool that also must be included.

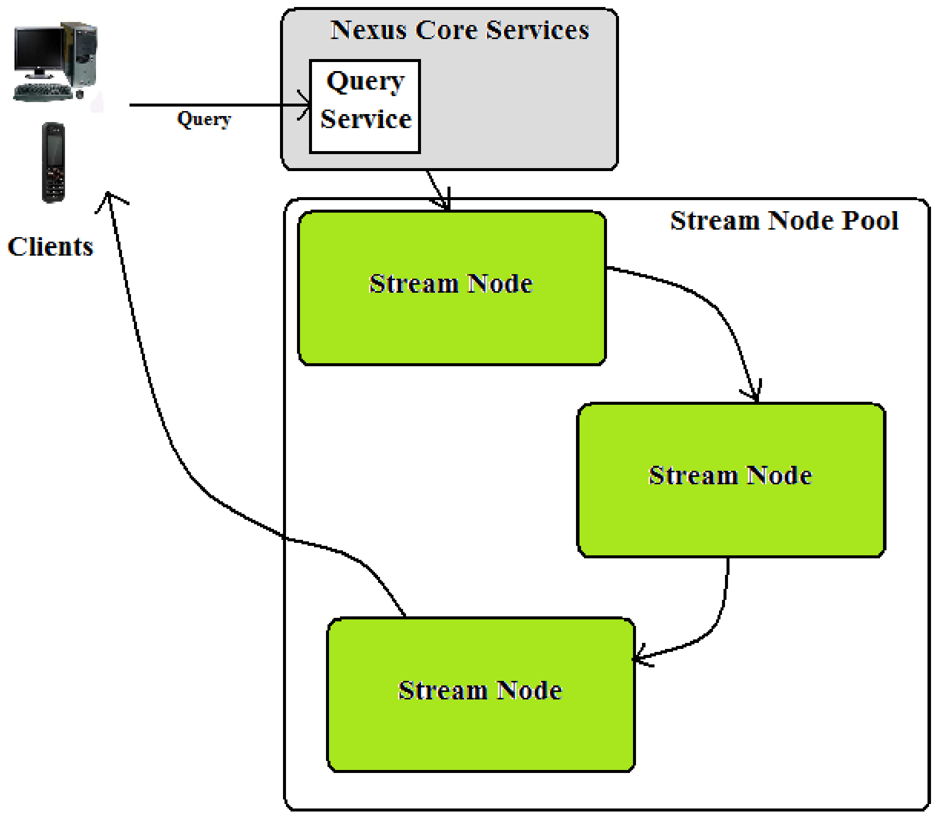


Figure 4.2: Relation between NexusDS core components a the Stream Node Pool

Most of the architecture elements of the Stream Platform and Stream Node Pool were conserved from the design of [10] in order to maintain a homogeneous architecture across platforms; besides that it is a proven architecture and is currently working.

4.3 Stream Platform

4.3.1 Objectives

Among others, the objective of the Stream Platform is to provide support to carry out the following operations.

- Loading a particular Stream Platform Back-End and setting it up.
- Loading the Root Network Group and joining it.
- Asking groups to find the Network Group Description of one of its children and joining it.
- Asking the group or groups where they are joined and finding other nodes in the network for Node Descriptions.
- Sometimes the description stores a Connection ID that can be used to create a connection within the node owner of that description.

- Having the Connection ID, the node can ask the group for a client connection, that can be either a Pipe (package based) or a Socket (flow based). If there is a Server Pipe or a Server Socket open with the same Connection ID in a server node, the connection will be established.
- Asking the group for a new Connection ID, and using it to create a new Server Pipe or Server Socket.
- Publishing its Node Description into the group containing this Connection ID.
- Closing sockets, leaving groups and asking the platform to stop running.

The different entities mentioned here will be further explained in the following points.

4.3.2 Architecture

There are two principal components. The first one is the Stream Platform Front-End that is composed by a set of factories and interfaces that an application will use to access the network and the Stream Nodes in our framework. The second one is the Stream Platform Back-End, that is the current implementation of the front- end, and will interact with a determined connectivity platform in order to reach the network [see 4.3].

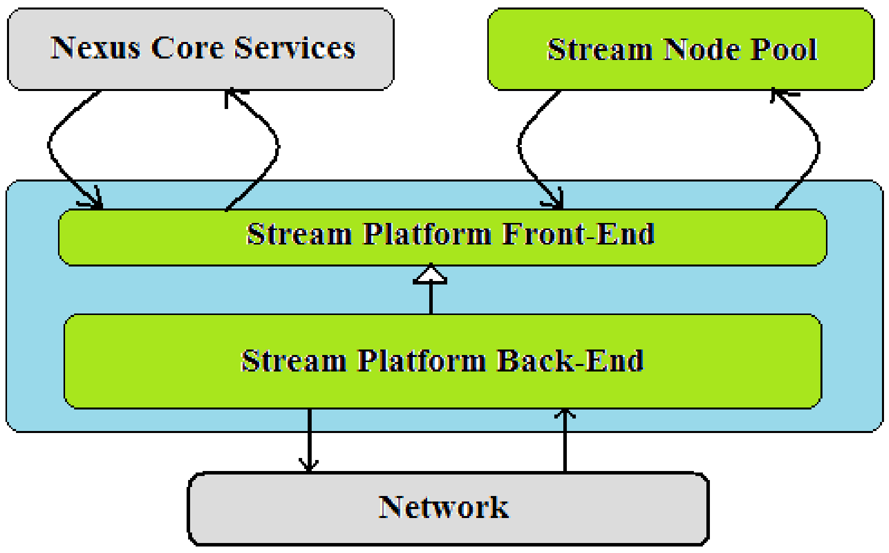


Figure 4.3: Stream Platform Front-End and Stream Platform Back-End

This design allows the isolation of the network and tries to be flexible, thus allowing having several back end implementations for different connectivity platforms.

In the next points, the different elements of the Stream Node design will be explained and at the end, a small use case is described in order to understand the functionalities of each element better.

4.3.3 Components

Network groups

The nodes will be organized in Network Groups. This logical partition can be made regarding the nodes services, their non-changing characteristics, their restrictions, their purposes, kind of device or even geographic situations. Each peer belongs to one or more groups at the same time, therefore these groups can overlap. Also, any node is able to create, join, leave or find a group.This approach helps to achieve a more flexible system and enables a more practical control of resources and nodes.

This structure is hierarchical since a group has only one ancestor (being the Root Network Group the root of the tree). This way, each group provides the functionality to look for its parent or its registered children. The organization of the nodes in a hierarchical way provides the search for the most adequate node in the network to carry out a specific subtask at a particular point in time.

Every group of the Stream Platform will be represented by a Network Group Description, a data structure used to summarize the most important characteristics of a Network Group. A Network Group Description is required to create a new group, or instantiating and joining a preexisting one.

Four main groups are defined in our system:

Root Network Group It is the default group for the platform, and the root of the group hierarchy. It may be shared with others applications and services around the world, and will not isolate our system from the rest. This group should only be used to initially bind to the network and look for the Default Nexus Group.

Default Nexus Group It is the default group for any node on NexusDS using the Stream Platform. This group's purpose is to be the parent of all the groups that are associated to the NexusDS platform, and to allow a node to find and join those groups. As an example, all nodes that form the Nexus Stream Federation should be together in the Stream Node Group, a child of the Default Nexus Group.

Stream Node Group All the Stream Nodes in the network should join this group. It will be the one used to make the connections between the Stream Nodes and also to send initialization, modification or cancelation commands to these nodes.

Stream Mobile Node Group Child of Stream Node Group, this group is a specific group for the Stream Nodes Running in mobile devices. These nodes should join this group, and as the Stream Node Group, organized in hierarchical structure depending of properties like display size, memory and processor capabilities, and many elements that differentiate the wide range of different mobile devices.

Network services

The Network Services are out of the scope of this thesis but the principal idea is explained in this point to be taken into account in the system.

A Network Service is an application running on top of a peer, or distributed among several peers ,that offers some kind of functionality to other peers in a Network Group. In order to do so, the peer or peers offering this service will need to open a server connection, and the client peers will need to open client connections to any of this server peers by using the information stored in a Network Service Description.

Network Service Description is used to describe the service itself, regardless of which nodes are providing it. The benefits of publishing a Network Service Description, instead of directly creating a Server Pipe with a well known Connection ID, lies on the possibility for other peers in the network to dynamically discover a service by knowing its name, id or functionality description, and then be able to create a client Pipe. In point 4.1 there are some examples of Network Services: Monitoring Services, Command Repository Services and Query Services.

Messages

A Message is a set of named and typed pieces of information that consists in the basic unit of data exchange to be transmitted through a Pipe. Messages wrap every piece of data that need to be sent, regardless of its nature. A single Message is able to store several data elements of the same or different data type that are stored and retrieved using keywords. The only limit is the Maximum Transfer Unit (MTU) that imposes an upper bound to the size of the data to be transmitted. For this reason, if the information that needs to be transmitted is bigger than the MTU, it will need to be split into several messages, with the knowledge needed to join them at the receiver side.

In some occasions it is more convenient to avoid splitting the information in chunks and making the required mechanisms to reorder the pieces and put them together.

Pipes and Sockets

There are two different ways to carry out communication between nodes: Pipes and sockets. Both need the concept of connection ID.

Connection ID Connection IDs refer to a connection, regardless of whether this connection is a Pipe or a Socket. This identifier will uniquely determine a connection within the context of the whole Stream Platform. Whenever a new connection needs to be created, both the server and the client parts need to use the same Connection ID.

A Connection ID is generated by a Network Group instance to which it will be associated. A Connection ID created by a particular group may not be valid to obtain a connection on a different group, and even if it were, this practice is strongly discouraged.

When a client node opens a client connection (Pipe or Socket) to a specific Connection ID it can determine the Node ID of the only node that is allowed to answer the connection request. This functionality is needed if the connection must be answered for a specific node.

A pipe is an abstraction used by the peers in the Stream Platform in order to communicate. Pipes are bidirectional and reliable communication channels that are used by the nodes on the Stream Platform to transfer information with each other. Both unicast and multicast pipes are available.

The connectivity offered by a Pipe is packet based. This means that the information should be split into Messages whose size is less than or equal to the specified. Maximum Transfer Unit (MTU). This limit is imposed by the platform back-end implementation, and needs to be documented.

In this context, saying that a Pipe is reliable means that if a Message is sent without a problem, it will be received on the other Pipe end. It is not ensured that Messages will arrive in the proper order that they were sent. Restoring this order should be done at the application level, if it is needed.

For this reason, this kind of connection will perform better for sending multiple short pieces of information, that are independent of each other, than for sending a single big one. If a big long piece of information should be transmitted, a Socket should be preferred to a Pipe, because Sockets guarantee that, if the information is successfully sent, it will be received in the same order that it was sent.

A pipe connection is uniquely identified by its Connection ID. If two nodes in the same group want to establish a connection between them, both of them should know this ID. Here we can have several cases, regarding how both nodes get this identifier:

- A third party entity delivers the Connection ID to both nodes.
- The server node lets the client node know about the chosen ID, usually by advertising it within a Network Service Description or a Node Description.

The server node will create a Server Pipe, and wait for incoming connections. Then, a client node will be able to instance a client Pipe with the same Connection ID. Note that several clients will be able to connect to the same Server Pipe at the same time. Moreover, several Server Pipes can be opened at the same time in different nodes with the same Connection ID. This is done this way in order to permit service replication. If a connection to a particular peer is needed, then its Node ID should be specified when creating the client Pipe. One example of use is for sending commands from one node to another one.

A socket is an abstraction, similar to a Pipe that allows information to be sent and received between two nodes. A Socket is created when a node opens a Server Socket in a specific group with a determined Connection ID and a connection request arrives from a client node.

Sockets do not use Messages to communicate. The information is continuously written as raw data on a Socket endpoint which is entrusted to split it up, and to reorder it in the other endpoint when necessary. The type of the data sent through a Socket cannot be specified, as it can be done with Messages.

Note that a Socket is a unicast connection; it cannot link more than two peers. Also, it cannot be used to provide a Network Service.

One example of use is for sending a big piece of data like a photo form one node to another one.

Network Nodes

The network nodes will be defined and identified in the network by both Node Description and Node Dynamic Description that are used by a node to publish its characteristics on a Network Group. E.g. a Stream Node will publish both descriptions on the Stream Node Group. This way, it is able to determine if a Stream Node has the needed profile to run an specific subtask.

Node ID A Node ID will uniquely identify a node inside a Network Group. It is not required that a peer have the same ID in different groups. The ID cannot be arbitrarily chosen: it will be given by the Stream Platform Back-End when a request is made to a Network Group instance. This ID will be stored by the platform in a stable memory, and will be retrieved the next time the platform starts.

Node Description In order to know about the hardware and software capabilities of a node, we will have to look for a Node Description. This will be used to check if one specific node covers the specifications we need to carry on with a specific task. Including the Node ID, the Node Description only contains those properties that will not change frequently. Some properties will be the name of the node, its pipe advertisement, information of its Operating System, its hardware architecture, total size of the physical RAM and network information like its IP.

Node Dynamic Description If we want to specify more dynamic properties like the CPU load, or the java application memory load, we should create a Node Dynamic Description. One mandatory property will be the Node ID that will allow a node to be recognized within a network group.

As this description changes very frequently, publishing this description in a Network Group within a very short period of time could be an inefficient strategy. This Node Dynamic Description should be published when there is a big change in one of the parameters with respect with the last published description, or when it is requested by the Resource Manager.

Securely identifying a node within the Stream Platform will only be possible by using a X.509 certificate. The public part of a X.509 certificate (see [11]) could be added to this description.

4.4 Stream Node Pool

This is the component that is more explained, core of the framework and that will be implemented in the prototype. Once again, almost of the next points are based in the previous design work of [10]. There are some variants adapting some points to the new mobile devices platforms.

4.4.1 Objectives

Basically, the main objective of this component is compute all the subtasks through the operations implemented in each node, therefore, a stream processing will be made by the pool of nodes.

Among others, every single Stream Node in the pool must be capable, at least, of the following functionality:

- Receive commands from the Resource Controller and give some feedback regarding their success.
- Parse a received command, and if it is sintactically and semantically correct, take the appropriate measures.
- Host the Process Line that will make the stream processing.

4.4.2 Architecture

The Stream Node Pool is composed of nodes called Stream Nodes, and they are organized into several Network Groups in order to allow wide scalability, being the Stream Node Group the root of this group hierarchy.

4.4.3 Components

Every node in the Stream Federation that makes part of the stream processing is called a Stream Node. This node is not more than a processing device that runs the Stream Node application. In a normal operation, only one instance will be running in one computer, and for this reason both the application and the computer hosting will be indiscriminately referred to as Stream Nodes within this document.

Figure 4.4 shows that a Stream Node consists of several parts:

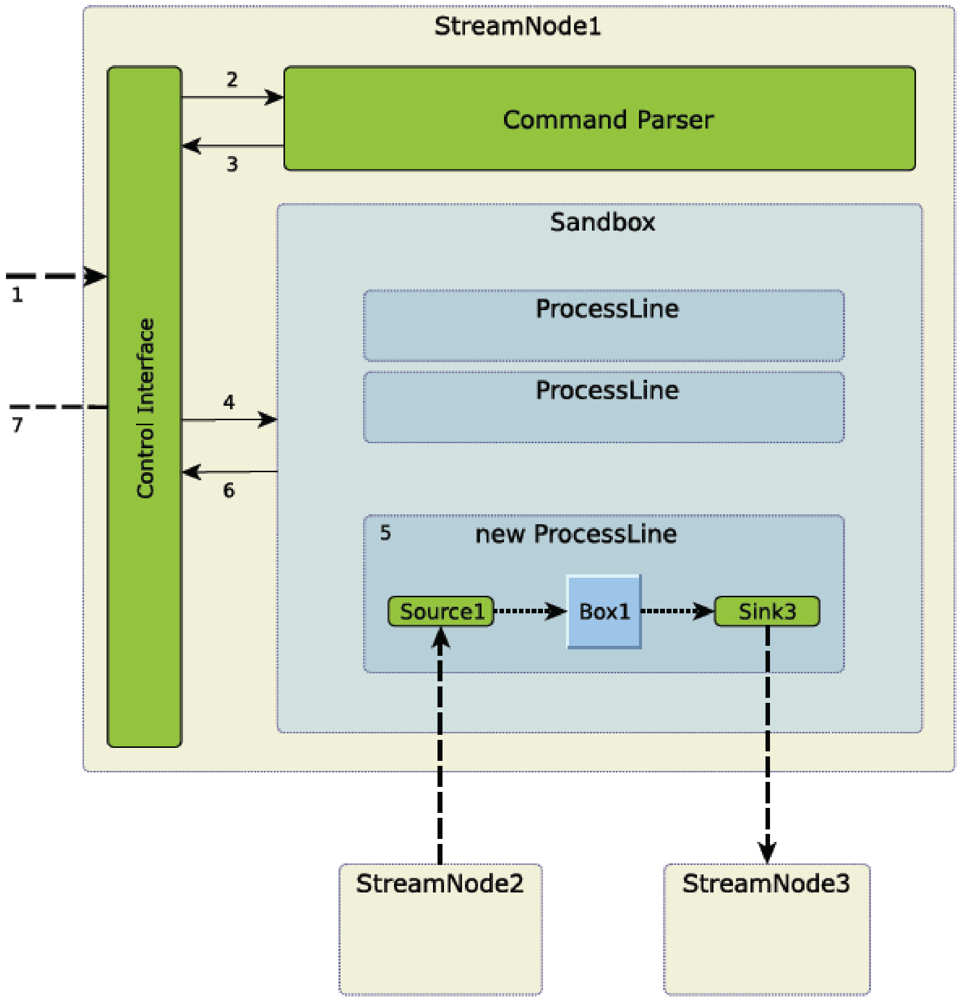


Figure 4.4: Stream node

`Stream Node Control` Interface receives control commands from the Resource Controller using the connectivity resources provided by the Stream Platform. It will return some feedback to the Resource Controller after the commands are processed.

`Command Parser` obtains the raw commands from the Control Interface, checks them syntactically, and gives them back as a complex tree data structure.

`Sandbox` gets the tree data structure produced and checks if the command it represents is semantically correct. The result of a successful command can be either to create, stop or modify a Process Line.

`Process Lines` are in charge of getting the actual stream processing made. Inside a Process Line, the input stream of data is obtained from the Data Sources. These inputs are processed by a line of Boxes, and their resulting output data streams are sent to the Data Sinks.

`Box` is the minimum complete stream processing entity. Each box includes one active component, the Input Manager and several passive components as Blocking Queues and Operations.

As the design of the Stream Node is a quite complex, it will be explained in detail in the point 4.5.

4.5 Stream Node

4.5.1 Stream Node Control Interface

The Stream Node Control Interface is the responsible of using the Stream Platform to connect to some of the core components of NexusDS, acting as a mediator between it and the rest of the internal components of the Stream Node. Furthermore, it should be able to process several incoming connections at the same time.

This entity requires the Stream Node to have started the Stream Platform and to be registered to the well known Stream Node Group and maybe also to one or several of its children groups.

The next sequence of events will explain the role of this entity better:

1. The Stream Node Control Interface receives an incoming connection through the Stream Platform from the Resource Controller.
2. The raw data received from this connection is passed to the Command Parser.
3. The Command Parser will return either a complex data structure result of the parsing the received raw command or some kind of error message, if the command was not syntactically correct. In the case of an error, the next step will be the last one exposed here.
4. The complex data structure given by the Command Parser is then passed to the Sandbox.
5. The Sandbox will check the semantic validity of the parsed action and process it by creating, stopping or modifying a Process Line.
6. The Sandbox returns some feedback to the Control Interface regarding the success or fail of the command processing.
7. The feedback returned by the Sandbox is then sent to the Resource Controller in the form of an Status Message.

The commands are designed to be extensible. Currently, the commands supported and allowed by the Stream Node Control Interface are the next:

Setup a Process Line with the described Data Sources, Data Sinks and setup Operations.

Setdown Stops a specific Process Line.

Update Modifies the properties of a running Data Source, Data Sink or Operation inside a Process Line.

4.5.2 Command Parser

It is the component that carries out the task of analyzing the commands and gives as a result a syntactically correct tree data structure or an error message.

The input raw command will be first analyzed lexically against a general formal serialization description, and so a sequence of tokens is created. In a next step, the type of the command will be recognized, and the formal description of this command type will be loaded. Finally, the tokens will be syntactically analyzed against the loaded command type description, and then the tree data structure will be created.

4.5.3 Stream Node Sandbox

In the context of Computer Security, a Sandbox is a security mechanism used for safely running untested or untrusted programs. Typically, it provides a controlled set of resources for these guest programs to run in, and they disallow or restrict the ability to inspect the host system, the network access, etc.

For our design, the Stream Node Sandbox follows a similar concept. In this case, the programs to run will be the Process Lines, and their execution should be somehow controlled. The first design idea is to allow only one process line per sandbox. Once again, this is because of the limited capacities of the mobile devices. Anyway, we do not know how powerful these devices can become in the future, therefore, the sandbox will be designed to support several process lines if it is possible and is configured in this way.

The sandbox is entrusted to manage all the Process Lines that are running inside a StreamNode. This responsibility includes storing, starting and stopping the processes, as well as carrying out a semantical analysis of the structures created by the Command Parser that represent commands in order to understand the data.

Every time a setup order is correctly processed, a new Process Line will be created and stored in the sandbox with a newly created Process Line ID. This identifier is independent from the Stream Platform, and needs to be unique only in the context of a Stream Node. It will be given as a receipt to the entity that requested the setup action. This information will be needed when requesting any modify or setdown commands, in order to specify which of the process in the sandbox needs to be modified or stopped respectively.

Process Lines

The whole stream processing chain is split into modules to be distributed among several computers and other devices. Each one of these modules is called a Process Line and will be running inside a Stream Node.

As it is seen in Figure 4.5, a process line is composed by:

`Data Sources`: Produces the stream data inside the process line.

`Boxes`: Entities that process the stream data.

`Data Sinks`: Consume the data processed before.

All these elements must be linked forming a path that goes from the Data Sources, through the Boxes to the Data Sinks. As some Operations could require some feedback, cycles in the data path are allowed.
In addition to storing all these components, the Process Line will also be responsible of starting them, stopping them and forwarding new parameter values to them, whenever an update command is received.

- How the data arrives and how it leaves

Here we are talking about how the data arrives to the process line, i.e., how the Data Source will receive the data. What is different is how the data the process entity receives (Box) inside the process line. In the same way, when we talk about how the data leaves the process line, we are talking about what the Data Sink does with the data that it receives from the process entity (Box).

The Operation inside the Box, which is explained after, always receives the stream data from Data Sources and always sends the processed data to Data Sink. But how it arrives and leaves the data in a process line is not always in the same way.

One option can be that the data arrives from other process line. For this connection the Platform Data Sources and the Platform Data Sinks are used. A second option is for example, that the data arrives from a Sensor to the Data Source.

- How the data is processed.

The transforming entities are the Operations. Belonging to an operation, it is always an Input Manager and buffering structures like Blocking Queues. All these components are grouped together in the Process Line to form a minimum complete, independent, active and isolated data transformer entity called Box.

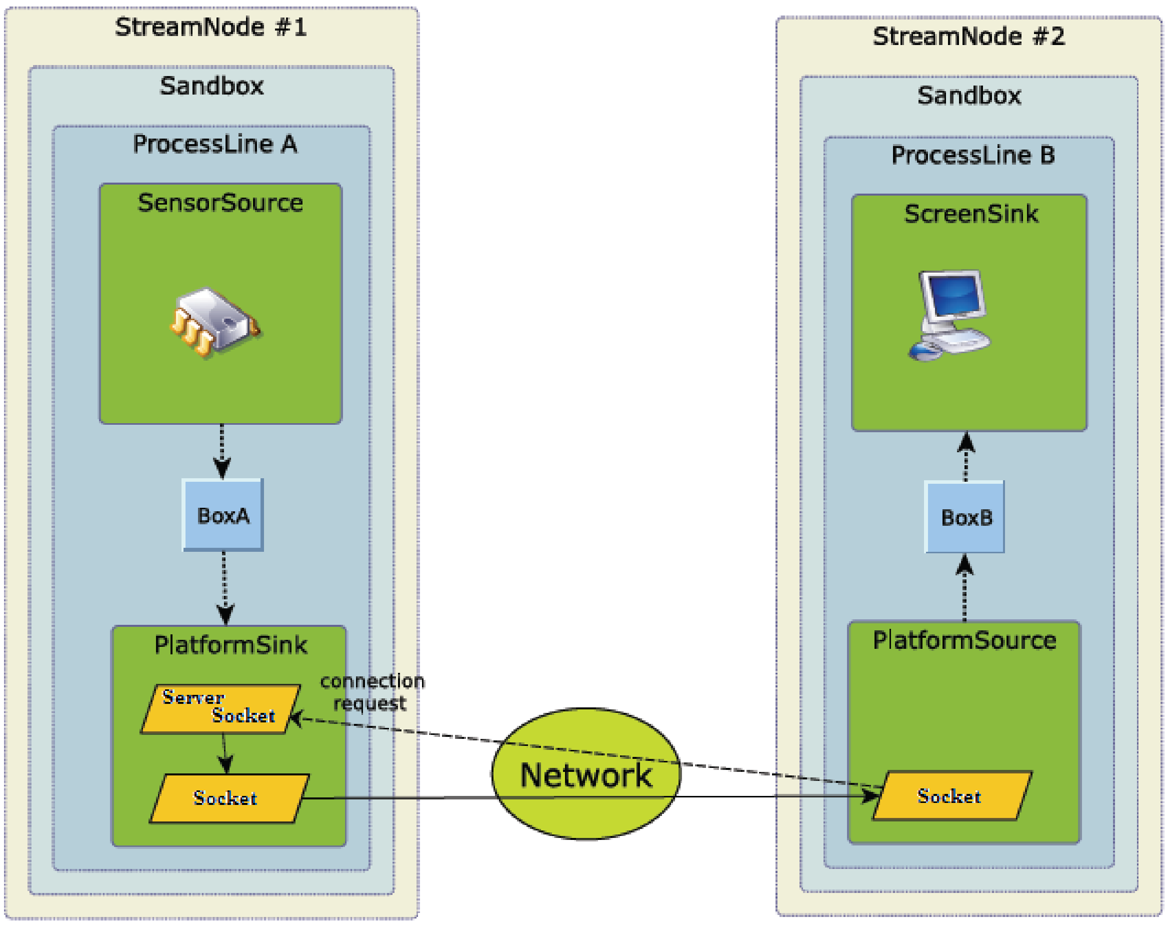


Figure 4.5: Communication between two process lines

Receivers

This is not an element in itself, but an abstraction that simplifies one as a receiver of data of a specific type. Each one of the outputs of a Box and a Data Source will have associated as many Receivers as needed, and each time the output produces some data, every one of them will be immediately notified. Valid Receivers can either be a Data Sink or an input of a box.

Data Sources and Data sinks

`Data Sources`

A Data Source is a component used inside a Process Line that asynchronously produces a stream of data. This component is an asynchronous and push-based producer of data. Due to these two properties, the interval of time between every data pushed to another element such a Box or Data sink is variable and each piece of data is pushed as soon as it is ready.

Inside the process line, the Data Source is seen like a data provider, but a Data Source really gets the data being an interface to a sensor or as an incoming link to other resources like Data Sinks located in different Process Line and different node. Therefore, the data unit and type can be very heterogeneous: numbers, strings, raw of bytes or more complex data. Each concrete implementation of Data Source must define the kind of data and cannot change at execution time.

The data is sent using only one output with the posibility of having many Receivers as needed. These receivers can be either Data Sink or the input if a Box.

One of the most important kinds of Data Source is the Platform Data Source. This enables getting data from other nodes through its Platform Data Sink. To achieve this remote connection, it is necessary to have information like the Network Group that has to be used and a Connection ID to establish the connection.

`Data Sinks`

Analogously to Data Sources, Data Sinks are also integrating components of a Process Line that asynchronously receive a stream of data. Unlike their counterparts, they cannot produce data or push data to another element. Like the Data Sources, a wide range of differents of data can be consumed in a Data Sink. Because of this, each implementation must define the kind of data that it will receive.

There are two different kinds of Data Sinks, the Platform Data Sinks and the Client Data Sinks.The Platform Data Sinks are imilarly to a Platform Data Source. It is a kind of Data Sink that will push data out of the Process Line through the Stream Platform. Also, this component will need to know which Network Group it needs to use and the Connection ID. With this knowledge, the Platform Data Sink will try to connect a Platform Data Source running on a remote computer's Process Line. The Client Data Sinks that can be employed for any other purpose. E.g. Store the data processed in the file system.

For the connection between a Platform Data Source and a Platform Data Sink could be either a Pipe or a Socket, being this last one the best approach due to its characteristics and behaviour described in the point 4.5. Both pipes and sockets need a server side and a client side, and the first needs to be started before the second. The Platform Data Sink will open the server side and the Platform Data Source will establish the connection with the role of client. In this way, several Platform Data Sources can connect to the server and this one will send data to every connection.

Boxes

This component can be considered as a complete, active and independent processing entity. As it is shown in Figure 4.6, each Box includes one Input Manager, one Operation, and as many Blocking Queues as input channels. This container will wrap all the inner components and act as a black box, so the Process Line does not need to know about its insides.

The responsibilities of a Box include:

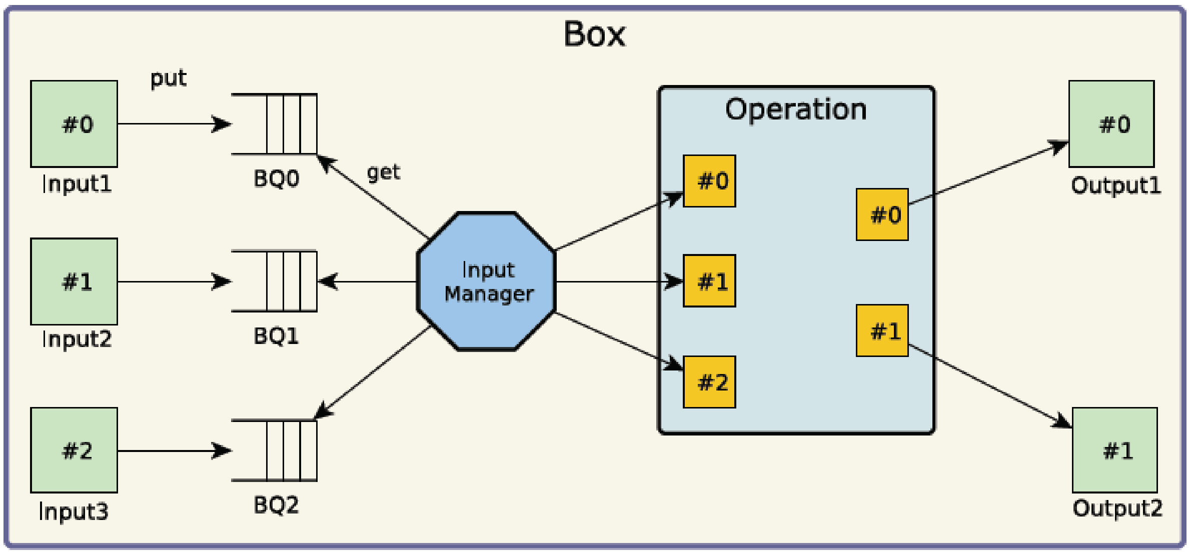


Figure 4.6: Internal components of a box

- Exposing the inputs and outputs of the inner Operation to the Process Line, so that they can be linked to other components, and checking that the mandatory ones are bound before the Box can start its processing.
- Checking that every input is associated to a Blocking Queue. If the setup command does not specify the type of queue that should be used, a default implementation of a Blocking Queue will be selected.
- Providing a default Input Manager if not requested differently by the setup command.
- Checking that all the mandatory outputs are bound, that means, at least one Receiver is associated to each one of them.

While interpreting the setup command, the Process Line creates a new Box each time a new Operation needs to be placed with its related Input Manager and Blocking Queues. But the Box itself is not described anywhere in the command content.

`Blocking Queues`: A queue is a special type of collection in which the data values are kept in order. The principal operations of this collection are the addition of values to the rear terminal position (add), and the removal of values from the front terminal position (remove). For this reason, it is said that a queue is a First-In- First-Out (FIFO) data structure [13]. The queues mainly perform the function as a buffer as they are mostly used to store and hold data values to be processed later. The main properties of a blocking queue give two behaviours:

1. When an element has to be retrieved and the queue is empty, a blocking queue offers a operation that will wait until an element is present.
2. When the queue is full and an element needs to be stored, a blocking queue offers an operation that wait until there is space.

In both cases, timeouts can be specified in order to avoid waiting indefinitely for storing or retrieving an element.

Queues could be positioned either at the input channels of a Box, at the output channels, or in both places. The decision to put the queues only at the input side came because it allows relinking Boxes without adding or deleting queues. If the queues were only at the output channels of a Box, it would be logical to have a different queue for each Receiver that is associated to the channel. Then, whenever a new Receiver would be added to the channel, a new queue would need to be created in front of it. On the other hand, when placing the queues at the input channels no new queues will be created inside the Box when it is relinked. The link from the channel to every Receiver will still be queued, but internally of the last. This means that a Box can be relinked across elements in the Process Line without actually creating new data structures inside of it. Also, in some applications where data loss is unthinkable, queues at both input and output channels could be a valid solution.

Some characteristics of the basic Blocking Queue can be modified in order to achieve other approaches. One property that can be changed is the fixed length, avoiding dynamic buffer size depending on different events, like overflow or free memory. Other approaches can be queues with other criteria than that of their arrival sequence or with replacing the elements depending on the importance in time. In short, several queues with different properties can be used according to the requirements or purpose. Again, allowing this variety of queues, the system gains flexibility and quality.

`Input Manager`: Is an active and reusable component that will control the work of an Operation pulling incoming data from the Blocking Queues and sending it to the Operation to make its process.

The idea is to have different Input Managers depending on the requirements of the Operation, therefore, the task of a Input Manager will be changed depending on its design characteristics. For example, if the operation needs several inputs, the input manager can have several threads to manage the data from every Blocking Queue, but it is possible that an operation only needs one input, so the input manager can be very simple in this case, avoiding several threads and saving resources, that in terms of mobile devices, is very important. A powerful capacity of customization of operations and their behaviour can be achieved with the idea of having an Input Manager Repository. We can think that not only one specific Input Manager is associated to a operation, allowing to select the one that is compatible with the operation and also is the best option attending to other requirements such as the capacities of the device where the operation is being executed.

When sending a setup command, a specific Input Manager can be selected from an Input Manager Repository. In other cases, a default Input Manager will be used.

Operations

An Operation is a passive component where a data transformation is done. This entity can receive this data through multiple inputs and give the results using multiple outputs. This internal processing is made accordingly to programmed instruction. Some elements that define an operation and that need to be specified are:

- Number of inputs, specifying which ones are mandatory and which are optional.
- Number of outputs, specifying which ones are mandatory and which are optional.
- Data type that must receive.
- Possible optional parameters to define properties.

This last point refers to another kind of information that some Operations also require in order to change the way they work. One example can be in an operation where some images are shown in the display of a mobile device. A parameter can establish how many images must be shown at the same time in the display. These properties are stored as key-value pairs inside the Operation and must have a default value.

Two approaches can be used to modify these properties dynamically at run time.

1. Using update commands. By this method, it is necessary to make some steps through different elements, starting in the Stream Node Control Interface that will receive the command, continuing with the Stream Node Sandbox and finalizing in the appropriate Process Line and its internal Box. This delegate chain may not be very efficient, so this option must not be used if the properties are continuously changing.
2. Specify an optional input to receive the properties values, contrary to the previous point, if The properties do not change frequently, it makes no sense spending resources in a new input (queue, possible new thread in input manger...etc).

As it has been said before, this element is passive. This means that it will not do anything by itself unless it has been ordered to do so. Therefore, an Operation needs to be associated to an Input Manager. This active and external element is independent and reusable. Then it was thought to place those two related components within a single Box, including as many Blocking Queues as needed (one per input channel of the Box). This way, all those dependent components were wrapped together into one, easier to set up, stop or destroy. For this reason, Boxes can be considered independent of each other because they have all they need to process the incoming data.

Regarding its participation in other core components of the NexusDS, each operation must have some kind of description in order to permit the Query Planner select suitable Operations and access them in the Operation Repository.

CHAPTER 5

PROTOTYPE IMPLEMENTATION

5.1 Introduction

A prototype was implemented following the system design, as a Proof of Concept. The two entities that are prototyped are the Stream Platform and the Stream Node Pool. Some points of the design are quite complex and are not included. In order to test the prototype, the Stream Node implementation will include some classes simulating actions in order to be capable of simulating and testing the capabilities of our framework developed.

In chapter 2 there is a description of the principal technologies that are used to carry out this implementation. J2ME as programming platform for mobile devices, JXME as network computing platform (over J2ME) and some development tools as Ant and Antenna for the MIDlet build process.

5.2 Project packets overview

This section is intended to give an overview of how it has translated the entities defined in the design phase to implementation. Thus, the reader or future developer is prepared for further explanation in more depth about the implementation of the most important elements. The prototype implementation is divided into eight projects and the next principal packets.

5.2.1 Stream Federation Packages

The contents of this package are all the interfaces and classes for the prototyping of the Stream Federation.It is structured in several subpackages which contents are explained below:

- `de.uni_stuttgart.nexus.streamFederation.node`: The main implementation of the Node Stream and the Node Stream Federation.
- `de.uni_stuttgart.nexus.streamFederation.sandbox`: The main implementation of the Sandbox and the Process Line.
- `de.uni_stuttgart.nexus.streamFederation.utils.command`: The definition of the Command interface, Command Elements interfaces and implementation of these interfaces, like a Setup Command and a Setdown command.
- `de.uni_stuttgart.nexus.streamFederation.utils.configuration`: The implementation of classes for the Configutarion of the Stream Node.
- `de.uni_stuttgart.nexus.streamFederation.boxes`: The definition of several interfaces for a Box, Operation, Input Manager, Receiver, Queue, Source and Sink among others. Also some abstract implementation of these interfaces.
- `de.uni_stuttgart.nexus.streamFederation.gui`: The definition of interfaces for the Controller, the Stream Node Facade, the Stream Node Launcher, View and MIDlet.
- `de.uni_stuttgart.nexus.streamFederation.inputManagers`: The implementation of Input Managers.

- `de.uni_stuttgart.nexus.streamFederation.operators`: The implementation of Operators.
- `de.uni_stuttgart.nexus.streamFederation.queues`: The implementation of Queues.
- `de.uni_stuttgart.nexus.streamFederation.sinks`: The implementation of Sinks.
- `de.uni_stuttgart.nexus.streamFederation.sources`: The implementation of Sources.
- `de.uni_stuttgart.nexus.streamFederation.utils.sandbox`: The implementation of Loaders for concrete Input Managers, Operations, Queues, Sinks and Sources.

5.2.2 Stream Platform Packages

The contents of this package are all the interfaces and classes for the prototyping of the Stream Platform. It is structured in several subpackages which contents are explained below:

- `de.uni_stuttgart.nexus.streamPlatform`: The definition of interfaces for the Network Group, Network Group Description, Network Group Factory, Platform andPlatform Factory.
- `de.uni_stuttgart.nexus.streamPlatform.pipes`: The definition of interfaces for Pipes.
- `de.uni_stuttgart.nexus.streamPlatform.services`: The definition of interfaces for the Node Description, Network Service Factory and Network Service Instance Description.
- `de.uni_stuttgart.nexus.streamPlatform.sockets`: The definition of interfaces for the Sockets.
- `de.uni_stuttgart.nexus.streamPlatform.jxtaBackend`: The implementation of the Network Group, Network Group Description, Network Group Factory, Discovery Service, Description Discovery Service and Platform.
- `de.uni_stuttgart.nexus.streamPlatform.jxtaBackend.configuration`: The implementation of classes for the configuration of the Platform.
- `de.uni_stuttgart.nexus.streamPlatform.jxtaBackend.pipes`: The implementation of Client and Server Pipe.
- `de.uni_stuttgart.nexus.streamPlatform.jxtaBackend.services`: The implementation of the NetworkServiceFactory, Network Service Instance Description, Node Description, and Node Description Advertisement.
- `de.uni_stuttgart.nexus.streamPlatform.jxtaBackend.sockets`: The implementation of the Client and Server Socket.

5.2.3 JXME packages

The contents of this package are all the interfaces and classes of the JXME API. It is structured in several subpackages that are not explained here because it is out of the focus of the implementation; only small changes over the implementation were made in order to fix the problems described in 2.5.4.

5.3 Stream Platform

As explained in 4.3 the Stream Platform is an Application Programming Interface (API), that isolates the Stream Federation and Stream Node applications from the used connectivity platform. This API uses the adapter and factory design patterns (see [8]) to wrap different disparate implementation back-ends preserving a unified coding interface. Each one of those back-ends will allow the use of a different connectivity subsystem. In this prototype, the JXTA Stream Platform Back-End is provided as JXTA was selected as the best choice for the purpose of this thesis.

The reader will be able to find a big similarity between the provided Stream Platform API, and the JXTA platform. This was done on purpose to take full ad vantage of the JXTA capabilities while trying to protect the

Stream Platform user from the height complexity of the P2P framework.

The Uniform Modelling Language (UML) diagram of the main classes and interfaces that compose the actual prototype of the Stream Platform Front-End can be seen in the Figure 5.1.

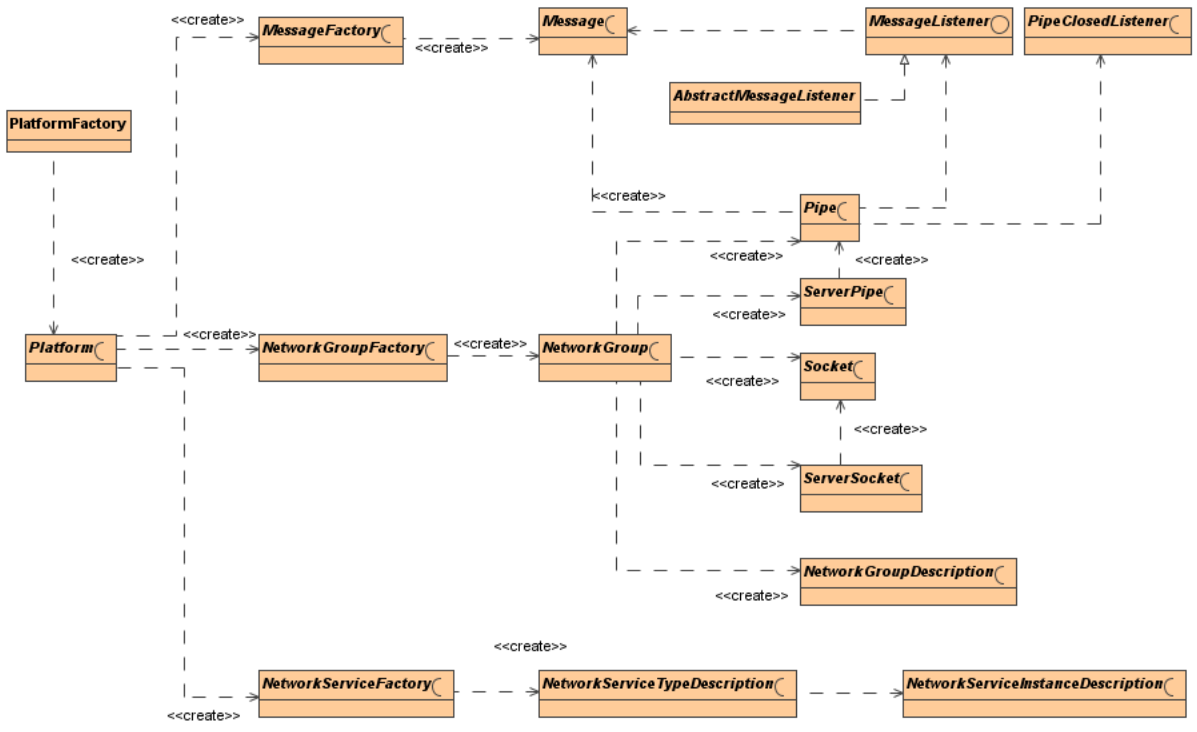


Figure 5.1: UML class diagram of the *Stream Platform Front-End*

5.3.1 Platform and Platform Factory

The interface Platform follows the design pattern Facade [8], where a single class provides an interface to accessing a more complex set of classes or system. In this case, the Platform provides all the factories and methods needed to access all the platform functionality (See Figure 5.2).

The Platform Factory provides a static method `getPlatform()` in order to load dynamically a specific Stream Platform Back-End indicating only the Java name of the class of this back-end that implements the Platform interface. The main methods for accessing the Stream Platform are:

- setupNetwork() : Configures some network parameters.
- startPlatform() : Initiates the Platform instance.
- stopPlatform() : Stops the Platform instance.
- getLocalNodeDescription(): Accesses the Node Description.

The factory offers methods for factories accessing which are:

- getNetworkGroupFactory(): creates an instance of the NetworkGroupFactory interface.
- getMessageFactory(): creates an instance of the MessageFactory interface.
- getNetworkServiceFactory(): creates an instance of the NetworkServiceFactory.

JXTA Back-End: JxtaBEPlatform

The JxtaBEPlatform is the class that implements the Platform interface in the JXTA Stream Platform Back-End. It is used to initialize the JXTA platform using static methods of the Configuration Factory to configure the JXTA platform and statics methods of PeerGroupFactory to start the platform. The relation between these classes can be seen in the Figure 5.3.

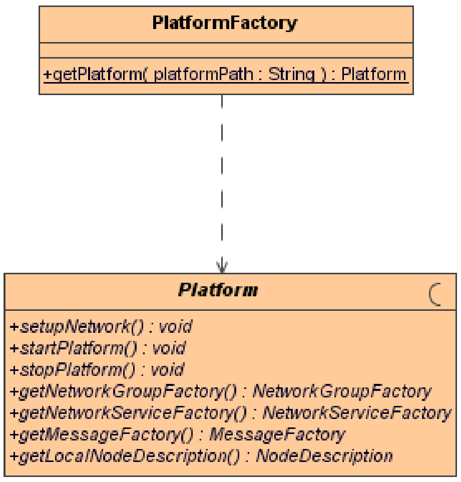

PlatformFactory
+getPlatform(platformPath : String) : Platform
Platform
+setupNetwork() : void
+startPlatform() : void
+stopPlatform() : void
+getNetworkGroupFactory() : NetworkGroupFactory
+getNetworkServiceFactory() : NetworkServiceFactory
+getMessageFactory() : MessageFactory
+getLocalNodeDescription() : NodeDescription

Figure 5.2: UML class diagram of *Platform and Platform Factory* interface.

When loading class JxtaBEPlatform, an instance of JxtaBEConfiguration is created to manage the JXTA configuration settings. After stopping the platform, the configuration, modified or not, is stored again in the non-volatile memory of the host platform.

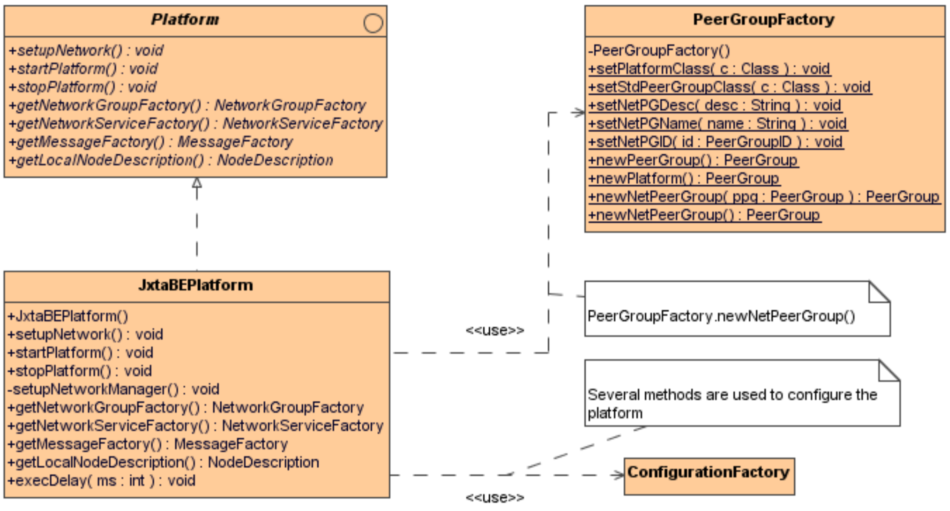

Platform
+setupNetwork() : void
+startPlatform() : void
+stopPlatform() : void
+getNetworkGroupFactory() : NetworkGroupFactory
+getNetworkServiceFactory() : NetworkServiceFactory
+getMessageFactory() : MessageFactory
+getLocalNodeDescription() : NodeDescription
PeerGroupFactory
-PeerGroupFactory()
+setPlatformClass(c : Class) : void
+setStdPeerGroupClass(c : Class) : void
+setNetPGDesc(desc : String) : void
+setNetPGName(name : String) : void
+setNetPGID(id : PeerGroupID) : void
+newPeerGroup() : PeerGroup
+newPlatform() : PeerGroup
+newNetPeerGroup(ppg : PeerGroup) : PeerGroup
+newNetPeerGroup() : PeerGroup
JxtaBEPlatform
+JxtaBEPlatform()
+setupNetwork() : void
+startPlatform() : void
+stopPlatform() : void
-setupNetworkManager() : void
+getNetworkGroupFactory() : NetworkGroupFactory
+getNetworkServiceFactory() : NetworkServiceFactory
+getMessageFactory() : MessageFactory
+getLocalNodeDescription() : NodeDescription
+execDelay(ms : int) : void
<<use>>
PeerGroupFactory.newNetPeerGroup()
Several methods are used to configure the platform
ConfigurationFactory
<<use>>

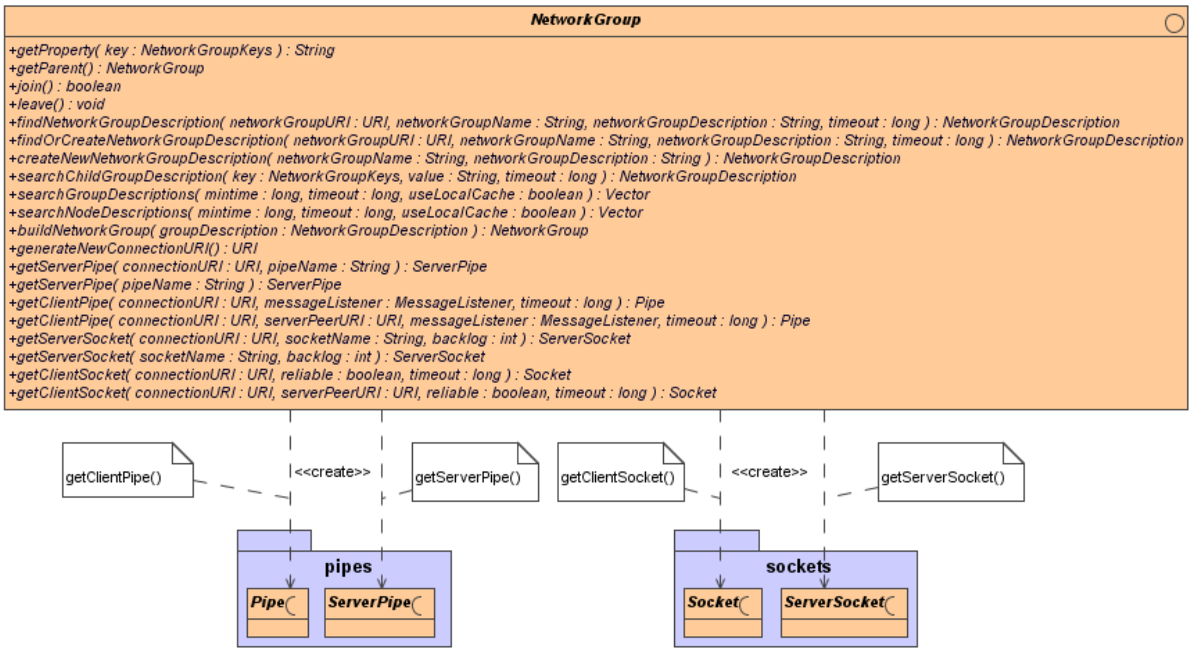
Figure 5.3: UML class diagram of the *JxtaBEPlatform* and its relation with *PeerGroupFactory* and *ConfigurationFactory* of the *Jxta API*.

The file configuration “jxtaConfig.txt” is stored in the folder “.sPlatformConfig/” in one of the roots of the mobile file system.

5.3.2 NetworkGroup and NetworkGroupFactory

The Network Groups form a hierarchy, where the Root Network Group is the root of the tree. The NetworkGroup-Factory provides methods to access the different groups of the hierarchy.

- getRootNetworkGroup()
- getDefaultNexusGroup()



```
+getProperty( key : NetworkGroupKeys ) : String
+getParent() : NetworkGroup
+join() : boolean
+leave() : void
+findNetworkGroupDescription( networkGroupURI : URI, networkGroupName : String, networkGroupDescription : String, timeout : long ) : NetworkGroupDescription
+findOrCreateNetworkGroupDescription( networkGroupURI : URI, networkGroupName : String, networkGroupDescription : String, timeout : long ) : NetworkGroupDescription
+createNewNetworkGroupDescription( networkGroupName : String, networkGroupDescription : String ) : NetworkGroupDescription
+searchChildGroupDescription( key : NetworkGroupKeys, value : String, timeout : long ) : NetworkGroupDescription
+searchGroupDescriptions( mintime : long, timeout : long, useLocalCache : boolean ) : Vector
+searchNodeDescriptions( mintime : long, timeout : long, useLocalCache : boolean ) : Vector
+buildNetworkGroup( groupDescription : NetworkGroupDescription ) : NetworkGroup
+generateNewConnectionURI() : URI
+getServerPipe( connectionURI : URI, pipeName : String ) : ServerPipe
+getServerPipe( pipeName : String ) : ServerPipe
+getClientPipe( connectionURI : URI, messageListener : MessageListener, timeout : long ) : Pipe
+getClientPipe( connectionURI : URI, serverPeerURI : URI, messageListener : MessageListener, timeout : long ) : Pipe
+getServerSocket( connectionURI : URI, socketName : String, backlog : int ) : ServerSocket
+getServerSocket( socketName : String, backlog : int ) : ServerSocket
+getClientSocket( connectionURI : URI, reliable : boolean, timeout : long ) : Socket
+getClientSocket( connectionURI : URI, serverPeerURI : URI, reliable : boolean, timeout : long ) : Socket
```

Figure 5.4: UML class diagram of the interfaces *NetworkGroup*, *Pipe*, *ServerPipe*, *Socket* and *ServerSocket*.

NetworkGroup is an interface that provides methods mainly for creating connections between nodes. The platform allows two different kinds of connections: Pipes and Sockets. Pipes need data to be organized in Messages before it can be transmitted, while Sockets send the information as a raw chain of bytes. Both kinds of connections can be obtained from an instance of NetworkGroup as shown in Figure 5.4. Also provides Connections ID.

A new group is always created or accessed using the methods provided by the instance of another group that is its parent. Therefore, an application using the Platform must first join a group and then will be able to use the methods of this group for different purposes.

In order to create a new NetworkGroupDescription, create `NewNetworkGroupDescription()` method of the parent is called , as arguments two strings with the name and a brief description of the new group.

With a NetworkGroupDescription it is possible to either look for the NetworkGroup that represents the child of the parent inside or create a new group that will be a new group child of the parent. The method used for this purpose is `BuildNetworkGroup()`.

Whenever an application wants to use a different group from Root Network Group or Default Nexus Group, it must join the group by using method `join()` on its instance. Similarly, if the application wants to leave a group, it must use method `leave()`.

JXTA Back-End: JxtaBENetworkGroup

Figure 5.5 shows that class JxtaBENetworkGroup is an adapter class that implements NetworkGroup and wraps the PeerGroup class of JXTA and its provided JXTA services as the DiscoveryService, the RendezVousService, etc.

NetworkGroupDescription

It is also shown in 5.5 that JxtaBENetworkGroupDescription is another adapter class. It implements Network-GroupDescription by wrapping PeerGroupAdvertisement which is used by JXTA to describe a PeerGroup.

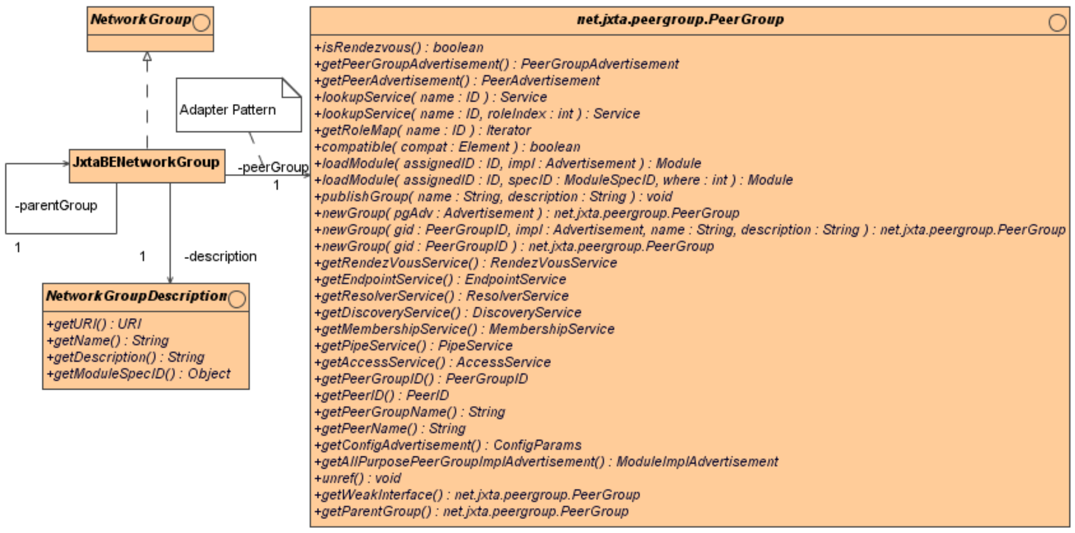

NetworkGroup
Adapter Pattern
JxtaBENetworkGroup
-peerGroup
1
-parentGroup
1
1
-description
NetworkGroupDescription
+getURI() : URI
+getName() : String
+getDescription() : String
+getModuleSpecID() : Object
net.jxta.peergroup.PeerGroup
+isRendezvous() : boolean
+getPeerGroupAdvertisement() : PeerGroupAdvertisement
+getPeerAdvertisement() : PeerAdvertisement
+lookupService(name : ID) : Service
+lookupService(name : ID, roleIndex : int) : Service
+getRoleMap(name : ID) : Iterator
+compatible(compat : Element) : boolean
+loadModule(assignedID : ID, impl : Advertisement) : Module
+loadModule(assignedID : ID, specID : ModuleSpecID, where : int) : Module
+publishGroup(name : String, description : String) : void
+newGroup(pgAdv : Advertisement) : net.jxta.peergroup.PeerGroup
+newGroup(gid : PeerGroupID, impl : Advertisement, name : String, description : String) : net.jxta.peergroup.PeerGroup
+newGroup(gid : PeerGroupID) : net.jxta.peergroup.PeerGroup
+getRendezVousService() : RendezVousService
+getEndpointService() : EndpointService
+getResolverService() : ResolverService
+getDiscoveryService() : DiscoveryService
+getMembershipService() : MembershipService
+getPipeService() : PipeService
+getAccessService() : AccessService
+getPeerGroupID() : PeerGroupID
+getPeerID() : PeerID
+getPeerGroupName() : String
+getPeerName() : String
+getConfigAdvertisement() : ConfigParams
+getAllPurposePeerGroupImplAdvertisement() : ModuleImplAdvertisement
+unref() : void
+getWeakInterface() : net.jxta.peergroup.PeerGroup
+getParentGroup() : net.jxta.peergroup.PeerGroup

Figure 5.5: UML class diagram of the *JxtaBENetworkGroup* and the adapted class *PeerGroup* of the *Jxta API*.

Whenever it is required to create a new PeerGroup, or load a preexisting one, two objects are needed:

- The instance of the parent PeerGroup.
- The PeerGroupAdvertisement that stores all the information regarding the new or preexisting PeerGroup, as its name, description, PeerID, etc.

In the JXTA API there is a coding convention that states that every class whose name ends with "Advertisement" (as PeerGroupAdvertisement does) should extend Advertisement, an abstract class used to represent JXTA resources. Nodes in a peer group will be able to find these advertisements if they are previously published. This can be done by one peer with the aid of the DiscoveryService associated to the PeerGroup instance.

JXTA Back-End: JxtaBENodeDescription

The JxtaBENodeDescription is an abstract class that:

- Implements the NodeDescription interface on the Stream Platform Front-End.
- Extends the ExtendableAdvertisement on the JXTA platform.

NodeDescriptionAdvertisement is a class that:

- Extends JxtaBENodeDescription.

NetworkGroupDescription

The facilities offered by JXTA are used here in order to create custom Advertisements and publish them. To achieve that, this complex structure is necessary, which will allow to create a Node Description and publish it over the platform. In this way, one peer can publish its description in a group and the other peers in the group can retrieve it.

But one step more is necessary to be able to achieve this functionality. After defining NodeDescriptionAdvertisement,it is necessary to register it in the AdvertisementFactory of JXTA to be able to publish, recognize and instantiate the NodeDescriptionAdvertisement, that now is another Advertisement more in the platform like for example a PipeAdvertisement. This registration is carried out in the class JxtaBEPlatform when the jxta platform in being configured.

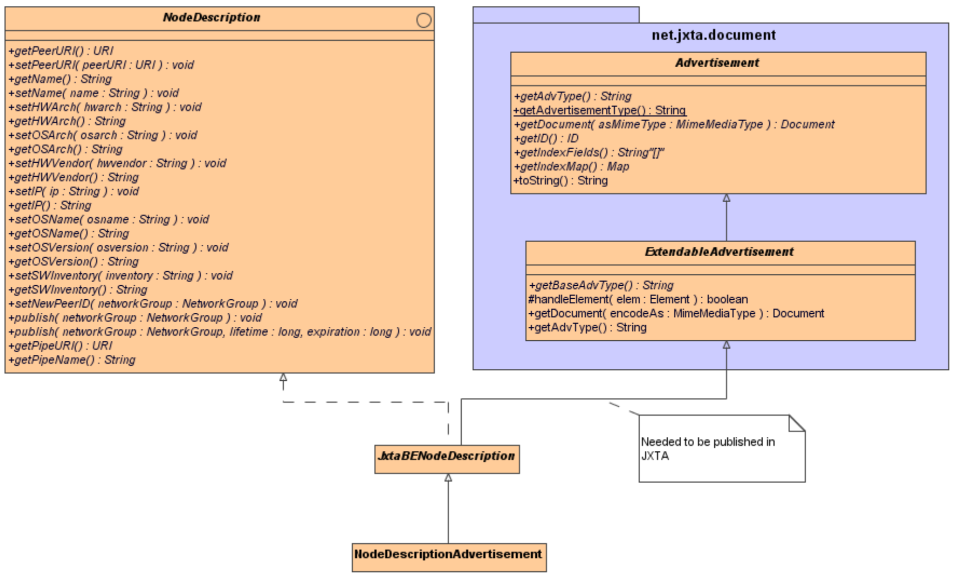

NodeDescription
+getPeerURI() : URI
+setPeerURI(peerURI : URI) : void
+getName() : String
+setName(name : String) : void
+setHWArch(hwarch : String) : void
+getHWArch() : String
+setOSArch(osarch : String) : void
+getOSArch() : String
+setHWVendor(hwvendor : String) : void
+getHWVendor() : String
+setIP(ip : String) : void
+getIP() : String
+setOSName(osname : String) : void
+getOSName() : String
+setOSVersion(osversion : String) : void
+getOSVersion() : String
+setSWInventory(inventory : String) : void
+getSWInventory() : String
+setNewPeerID(networkGroup : NetworkGroup) : void
+publish(networkGroup : NetworkGroup) : void
+publish(networkGroup : NetworkGroup, lifetime : long, expiration : long) : void
+getPipeURI() : URI
+getPipeName() : String
net.jxta.document
Advertisement
+getAdvType() : String
+getAdvertisementType() : String
+getDocument(asMimeType : MimeMediaType) : Document
+getID() : ID
+getIndexFields() : String"[]"
+getIndexMap() : Map
+toString() : String
ExtendableAdvertisement
+getBaseAdvType() : String
#handleElement(elem : Element) : boolean
+getDocument(encodeAs : MimeMediaType) : Document
+getAdvType() : String
Needed to be published in JXTA
JxtaBENodeDescription
NodeDescriptionAdvertisement

Figure 5.6: UML class diagram of *JxtaBENodeDescription* and its related classes

5.3.3 Socket and ServerSocket

J2ME does not provide either implementation or interface classes for both client socket or server socket. Therefore, we create our own interfaces defining the basic methods following the implementation offered by Java Socket (see 5.7) and Java Server Socket in the standard platform. Every Stream Platform Back-End may need to extend these classes to make use of its own connectivity platform. The methods that are defined in the client interface are:

- Method `getInputStream()` that returns an instance of class InputStream that permits reading data from the socket.
- Method `getOutputStream()` that returns an instance of OutputStream.
- Method `close()` that closes the socket connection if it exists.

The methods that are defined in the server interface are:

- Method `accept()` that waits until it receives a connection from a client side.
- Method `close()` that closes the socket connection if it exists.
- Method `setSoTimeout()` that establishes the timeout for connections.

Creating a Socket connection

As we know, a socket connection is carried out between a client side (Socket) and a server side (Server Socket). The way to instantiate a Socket or ServerSocket in the proposed API differs from the standard. Instead of using these class constructors, the adequate factory method from NetworkGroup should be called.

There are two versions of the NetworkGroup's method `getServerSocket()`:

- In the first one a Connection ID is specified as an instance of URI.
- In the second one the Connection ID is not supplied and a new one is created.

As was stated in 4.3.3, in both cases the client node needs this Connection ID to connect to the server socket created. Also, there are two versions of the NetworkGroup's method `getClientSocket()`:

- In the first one, only the Connection ID is specified. This information may not be sufficient because different nodes may open a ServerSocket using the same Connection ID. In this case any of these peers can accept the connection.
- In the second one the peer ID is also specified as a URI, so only one peer will accept the connection.

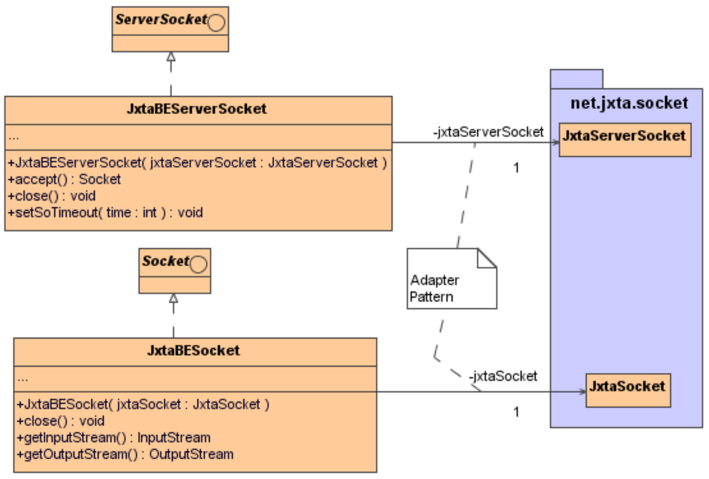

ServerSocket
JxtaBEServerSocket
...
+JxtaBEServerSocket(jxtaServerSocket : JxtaServerSocket)
+accept() : Socket
+close() : void
+setSoTimeout(time : int) : void
Socket
JxtaBESocket
...
+JxtaBESocket(jxtaSocket : JxtaSocket)
+close() : void
+getInputStream() : InputStream
+getOutputStream() : OutputStream
Adapter Pattern
-jxtaServerSocket
1
-jxtaSocket
1
net.jxta.socket
JxtaServerSocket
JxtaSocket

Figure 5.7: UML class diagram of *JxtaBESocket* and *JxtaBEServerSocket* together the adapted classes of *Jxta API*.

JXTA Back-End: JxtaBESocket and JxtaBEServerSocket

As seen in Figure 5.7, both JxtaBESocket and JxtaBEServerSocket are adapter classes that implement Socket and ServerSocket respectively.

JxtaBESocket wraps JXTA's JxtaSocket and JxtaBEServerSocket wraps JXTA's ServerSocket. These classes of jxta behave very much like Sockets, but really are a bdirectional Pipe. Each class defines its own protocol and the request that it receives are message requests.

[1] states that the JxtaSockets do "not implement Nagle's algorithm", therefore streams must be flushed as needed. This means that, whenever some data is written on the OutputStream provided by a JxtaSocket, it will not be sent through the network until method `flush()` from the OutputStream is invoked.

5.3.4 Pipe and ServerPipe

On a different approach to sending streams of raw data through the network, the Pipe paradigm is proposed. Before being sent, the data is organized in typed pieces of information that is stored in individual Messages. The prototype only privides unicast pipe, with realiable and unreliable options.

The Messages received by an instance of class Pipe can be obtained by the application in two different ways:

1. The application invokes Pipe's method `receiveMessage()` that will block till a Message is received or the timeout passed as an argument expires.
2. The application implements interface MessageListener whose only method `messageReceived()` will return a Message as soon as it is received by the Pipe.

This second way to receive Messages from a Pipe is preferred since using method `receiveMessage()` requires the Pipe to have an internal queue and messages can be lost if this queue overloads. This first way only works if no MessageListener is registered on the Pipe. Also, a Pipe can inform the application that the connection is closed by the remote node, or that it is lost. This is made by registering the application itself on the Pipe as a PipeClosedListener.

Creating a Pipe connection

Pipes work in a similar way to Sockets, therefore a server node opens a ServerPipe on a specific group and a client node opens a client Pipe on the same group with the same Connection ID. The way to instantiate a Pipe or ServerPipe is by using the adequate factory method from NetworkGroup.There are two versions of the NetworkGroup's method `getServerPipe()`:

- In the first one a Connection ID is specified as an instance of URI.
- In the second one the Connection ID is not supplied and a new one is created.

In both cases the client node needs this Connection ID to connect to the server socket created. Also there are two versions of the NetworkGroup's method `getClientPipe()`:

- In the first one, only the Connection ID is specified. This information may not be sufficient because different nodes may open a ServerPipe using the same Connection ID. In this case any of these peers can accept the connection.
- In the second one the peer ID as a URI is also specified, so only one peer will accept the connection.

JXTA Back-End: JxtaBEPipe and JxtaBEServerPipe

These classes provide bidirectional and reliable pipe implementations. In the JXTA platform, these properties are achieved with functionality given by JxtaServerPipe at the server side and JxtaBiDiPipe at the client side. For this reason, both JxtaBEServerPipe and JxtaBEPipe adapter classes are created on the JXTA Stream Platform Back-End. As it is shown in Figure 5.8, class JxtaBEServerPipe implements the ServerPipe interface by wrapping JxtaServerPipe, and JxtaBEPipe implements the Pipe interface by wrapping JxtaBiDiPipe.

As all the pipes in the JXTA platform, JxtaBiDiPipe can only send data that has been packaged in JXTA Messages. How the Stream Platform Messages are organized in those JXTA Messages is explained later in 5.3.5.

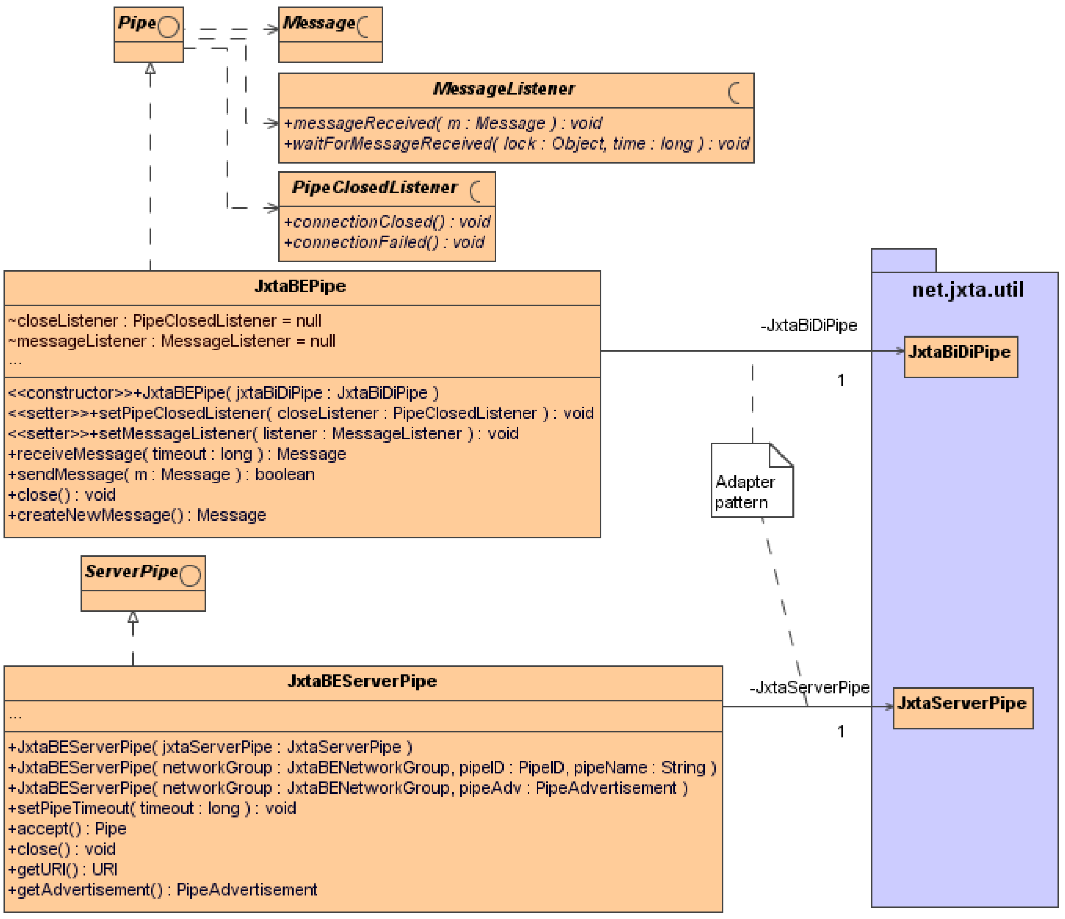

Pipe
Message
MessageListener
+messageReceived(m : Message) : void
+waitForMessageReceived(lock : Object, time : long) : void
PipeClosedListener
+connectionClosed() : void
+connectionFailed() : void
JxtaBEPipe
~closeListener : PipeClosedListener = null
~messageListener : MessageListener = null
...
<<constructor>>+JxtaBEPipe(jxtaBiDiPipe : JxtaBiDiPipe)
<<setter>>+setPipeClosedListener(closeListener : PipeClosedListener) : void
<<setter>>+setMessageListener(listener : MessageListener) : void
+receiveMessage(timeout : long) : Message
+sendMessage(m : Message) : boolean
+close() : void
+createNewMessage() : Message
net.jxta.util
-JxtaBiDiPipe
1
JxtaBiDiPipe
Adapter pattern
ServerPipe
JxtaBEServerPipe
...
+JxtaBEServerPipe(jxtaServerPipe : JxtaServerPipe)
+JxtaBEServerPipe(networkGroup : JxtaBENetworkGroup, pipeID : PipeID, pipeName : String)
+JxtaBEServerPipe(networkGroup : JxtaBENetworkGroup, pipeAdv : PipeAdvertisement)
+setPipeTimeout(timeout : long) : void
+accept() : Pipe
+close() : void
+getURI() : URI
+getAdvertisement() : PipeAdvertisement
-JxtaServerPipe
1
JxtaServerPipe

Figure 5.8: UML class diagram of *JxtaBEPipe* and *JxtaBEServerPipe* together the adapted classes of the *Jxta API*.

JxtaBiDiPipe is a high level pipe on the JXTA platform, and it is composed of two unidirectional simpler JXTA pipes.

5.3.5 Message

Every piece of information that wants to be transmitted through a Pipe must be stored within a Message. This data structure is modelled by the interface Message that offers several methods to store and retrieve different data types (see Figure 5.9). An instance of this class can store many pieces of data, each one identified by a key.

The Messages are used by Pipe, that provide the method `sendMessage()` to send messages.

The Stream Platform Back-End is responsible of serializing the message and checking by the method `getRemainingCapacity()` if the size of the message bytes exceeds the maximum size allowed by the communication platform.

JXTA Back-End: JxtaBEMessage

In this case, both the adapted and the adapted classes have the same name, as shown in Figure 5.9. To avoid ambiguities, the adaptee class belonging to the JXTA platform will be called jxta.Message in this text, aside from its real name which is Message. Having said that, it can be explained that JxtaBEMessage from the JXTA Stream Platform Back-End is an adapter class that wraps jxta.Message on the JXTA platform in order to implement class Message from the JXTA Stream Platform Front-End.

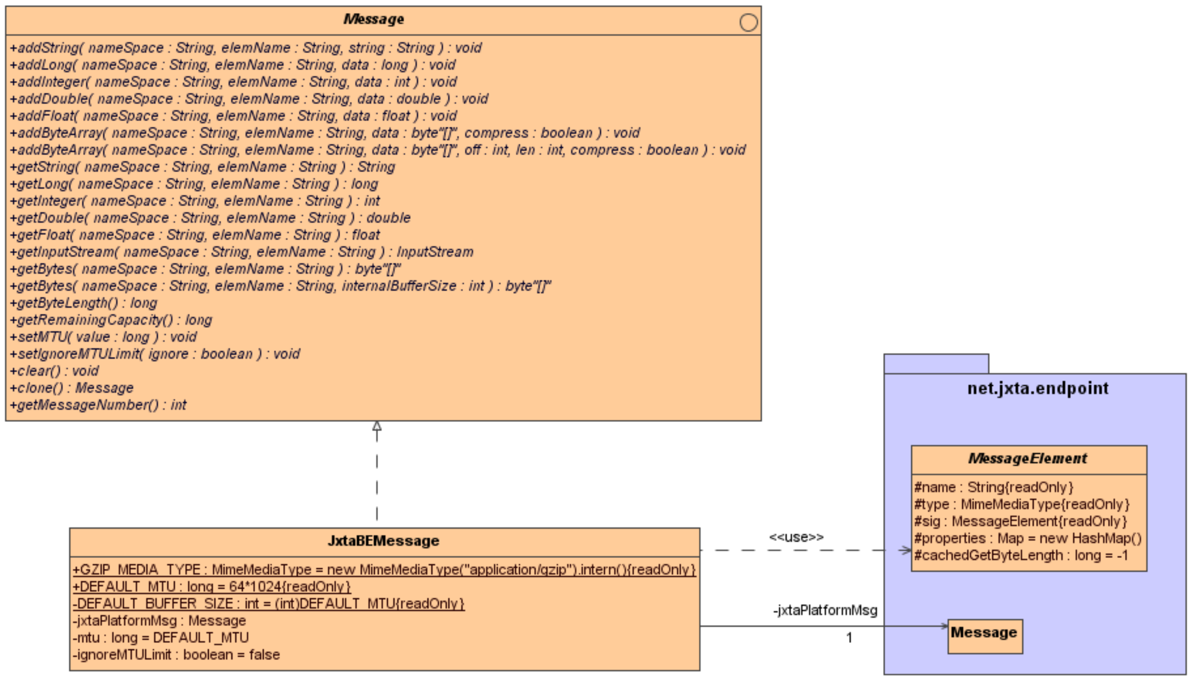

Message
+addString(nameSpace : String, elemName : String, string : String) : void
+addLong(nameSpace : String, elemName : String, data : long) : void
+addInteger(nameSpace : String, elemName : String, data : int) : void
+addDouble(nameSpace : String, elemName : String, data : double) : void
+addFloat(nameSpace : String, elemName : String, data : float) : void
+addByteArray(nameSpace : String, elemName : String, data : byte"[]", compress : boolean) : void
+addByteArray(nameSpace : String, elemName : String, data : byte"[]", off : int, len : int, compress : boolean) : void
+getString(nameSpace : String, elemName : String) : String
+getLong(nameSpace : String, elemName : String) : long
+getInteger(nameSpace : String, elemName : String) : int
+getDouble(nameSpace : String, elemName : String) : double
+getFloat(nameSpace : String, elemName : String) : float
+getInputStream(nameSpace : String, elemName : String) : InputStream
+getBytes(nameSpace : String, elemName : String) : byte"[]"
+getBytes(nameSpace : String, elemName : String, internalBufferSize : int) : byte"[]"
+getByteLength() : long
+getRemainingCapacity() : long
+setMTU(value : long) : void
+setIgnoreMTULimit(ignore : boolean) : void
+clear() : void
+clone() : Message
+getMessageNumber() : int
JxtaBEMessage
+GZIP_MEDIA_TYPE : MimeMediaType = new MimeMediaType("application/gzip").intern(){readOnly}
+DEFAULT_MTU : long = 64*1024{readOnly}
-DEFAULT_BUFFER_SIZE : int = (int)DEFAULT_MTU{readOnly}
-jxtaPlatformMsg : Message
-mtu : long = DEFAULT_MTU
-ignoreMTULimit : boolean = false
<<use>>
-jxtaPlatformMsg
1
net.jxta.endpoint
MessageElement
#name : String{readOnly}
#type : MimeMediaType{readOnly}
#sig : MessageElement{readOnly}
#properties : Map = new HashMap()
#cachedGetByteLength : long = -1
Message

Figure 5.9: UML class diagram of *JxtaBEMessage* and the adapted class *Message* of the *Jxta API*.

All the pieces of information that compose a jxta.Message should be stored as instances of one of the subclasses of MessageElement. Each one of these elements has a name associated to it, and they are stored in a jxta.Message using a namespace String. As a consequence, to identify one of these elements within a jxta.Message instance, both the namespace and the element name are required.

The adapter class JxtaBEMessage is in charge of selecting and instanciating the most fitting subclass of MessageElement when an adder method is invoked by an application. E.g. when method `addString()` is called on the adapter class with the String to be added as parameter, a new StringMessageElement will be created to store that string, and this new element will be added to the jxta.Message instance.

Similarly, when a getter method is invoked on JxtaBEMessage, this adapter will get from the wrapped jxta.Message the MessageElement corresponding to the provided namespace and element name. Then, the piece of data that this element stores will be obtained and returned to the application.

[1] stipulate that 64 KBytes is the maximum length of a serialized jxta.Message. This serialization includes both the data payload and the message headers. But the size of these message headers is not fixed, it varies with several factors, such as the quantity, type and name of the MessageElements added, the quantity and length of the namespaces key Strings, etc. Therefore the payload length also changes with these factors. For this reason method `getRemainingCapacity()` of class JxtaBEMessage calculates the remaining capacity of a particular JxtaBEMessage instance. This method will give an upper limit to the remaining payload length that could still be stored within that message.

The minimum overhead of a non empty jxta.Message serialization corresponds to the case when only one MessageElement of type ByteArrayMessageElement with the empty (“”) String as a name is stored in the message with the null String as a namespace. In this case, the size of the message header will be of 21 Bytes. Then, the maximum size of the byte array that a JxtaBEMessage can transport will correspond to 65536 Bytes-21 Bytes = 65515 Bytes.

If a piece of data of more than 65515 Bytes in length has to be transmitted through a Pipe, it should be split into several Messages on the application level. Also, reordering of the received Messages should be done on the receiver application. To avoid splitting and reordering, the application may consider using Sockets instead of Pipes.

5.3.6 NetworkService

Due to the fact that the Network Services are not needed for this prototype to work, their actual implementation will be left for future work.

5.4 Stream Node

The Stream Node application is represented by the StreamNode class. This will be responsible of loading and coordinating all the components needed to supply the Stream Node functionality that was described in 4.5. Figure 5.10 shows the relationship among the main implementation classes that constitute the Stream Node. Within this section, the most interesting details of the implementation of this prototype will be exposed, as well as some usage recommendations.

5.4.1 The StreamNode class

The StreamNode class is responsible ,among other things, of loading and setting up the Stream Platform, starting the Stream Node Control Interface and loading the Command Parser.

When the StreamNode is created, the first action is loading the configuration. One important value that can be obtained from the configuration is the Stream Platform Back-End full name.

The second step is loading a specific Stream Platform Back-End. At the present time there is only one choice: the JXTA Stream Plaform Back-End with its JxtaBEPlatform class.

Now, the Platform has to be started by calling `startStreamPlatform()`. This method is also responsible of obtaining the Net Peer Group (Root group) from the Platform instance. As was explained in 2.5.4, the current implementation of JXME does not allow the correct management of groups and it is only possible to access to the most basic peer group.

Only after having the Stream Node Group instantiated, the Stream Node Control Interface can be started by invoking `startStreamNode()`. After this point the computer running the StreamNode class is listening for Pipe connections on the Net Peer Group, and is able to process a setup command received through the pipe to create the first Process Line.

In order to shut the StreamNode down, methods `stopStreamNode()` and then `stopStreamPlatform()` must be invoked.

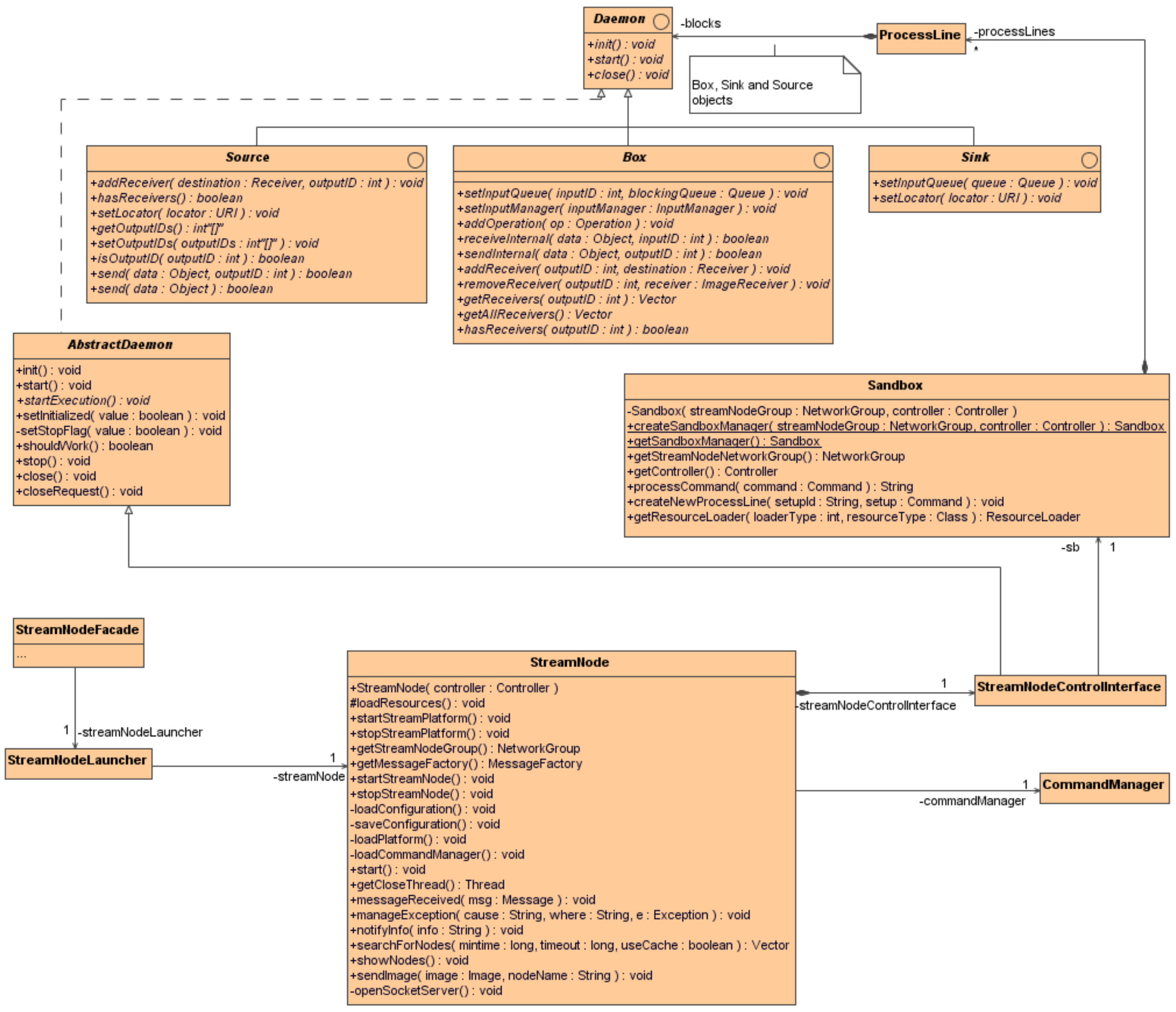

Daemon
+init() : void
+start() : void
+close() : void
-blocks
ProcessLine
-processLines
*
Box, Sink and Source objects
Source
+addReceiver(destination : Receiver, outputID : int) : void
+hasReceivers() : boolean
+setLocator(locator : URI) : void
+getOutputIDs() : int"[]"
+setOutputIDs(outputIDs : int"[]") : void
+isOutputID(outputID : int) : boolean
+send(data : Object, outputID : int) : boolean
+send(data : Object) : boolean
Box
+setInputQueue(inputID : int, blockingQueue : Queue) : void
+setInputManager(inputManager : InputManager) : void
+addOperation(op : Operation) : void
+receiveInternal(data : Object, inputID : int) : boolean
+sendInternal(data : Object, outputID : int) : boolean
+addReceiver(outputID : int, destination : Receiver) : void
+removeReceiver(outputID : int, receiver : ImageReceiver) : void
+getReceivers(outputID : int) : Vector
+getAllReceivers() : Vector
+hasReceivers(outputID : int) : boolean
Sink
+setInputQueue(queue : Queue) : void
+setLocator(locator : URI) : void
AbstractDaemon
+init() : void
+start() : void
+startExecution() : void
+setInitialized(value : boolean) : void
-setStopFlag(value : boolean) : void
+shouldWork() : boolean
+stop() : void
+close() : void
+closeRequest() : void
Sandbox
-Sandbox(streamNodeGroup : NetworkGroup, controller : Controller)
+createSandboxManager(streamNodeGroup : NetworkGroup, controller : Controller) : Sandbox
+getSandboxManager() : Sandbox
+getStreamNodeNetworkGroup() : NetworkGroup
+getController() : Controller
+processCommand(command : Command) : String
+createNewProcessLine(setupId : String, setup : Command) : void
+getResourceLoader(loaderType : int, resourceType : Class) : ResourceLoader
-sb
1
StreamNodeFacade
...
1
-streamNodeLauncher
StreamNodeLauncher
1
-streamNode
StreamNode
+StreamNode(controller : Controller)
#loadResources() : void
+startStreamPlatform() : void
+stopStreamPlatform() : void
+getStreamNodeGroup() : NetworkGroup
+getMessageFactory() : MessageFactory
+startStreamNode() : void
+stopStreamNode() : void
-loadConfiguration() : void
-saveConfiguration() : void
-loadPlatform() : void
-loadCommandManager() : void
+start() : void
+getCloseThread() : Thread
+messageReceived(msg : Message) : void
+manageException(cause : String, where : String, e : Exception) : void
+notifyInfo(info : String) : void
+searchForNodes(mintime : long, timeout : long, useCache : boolean) : Vector
+showNodes() : void
+sendImage(image : Image, nodeName : String) : void
-openSocketServer() : void
1
-streamNodeControlInterface
StreamNodeControlInterface
1
-commandManager
CommandManager

Figure 5.10: UML class diagram of the *Stream Node*

5.4.2 Stream Node Control Interface

As seen in Figure 5.10, the Stream Node Control Interface is implemented by class StreamNodeControInterface that indirectly implements the Daemon interface. This implies that this component is active, meaning that it runs on at least one Thread.

Loading the Stream Node Control Interface

The StreamNodeControlInterface requires the next entities to be created and start working: Platform, NetworkGroup, CommandManager and Controller. These references are passed as parameters in the constructor. Here, the main steps are made for configuring the interface:

1. Getting through the Platform the Node Description that, among others, contains the Connection ID and Peer ID.
2. Publishing the Node Description in the NetworkGroup, allowing the Stream Node to be found on that Network Group.
3. Creating and publishing in the NetworkGroup the Pipe Server used to receive the command.

Due to the fact that the Pipe Server and Node Description are published in the specific NetworkGroup, the StreamNodeControlInterface is bound to it and it is not possible to be bound to another Network Group. In the case that a Stream Node should receive commands from nodes in two different groups, two different instances of StreamNodeControlInterface should be loaded. In any case, all the possible instances of this class use the same Sandbox, as only one Sandbox instance is allowed on each StreamNode.

Initialization and starting the Stream Node Control Interface

After the Stream Node Control Interface is loaded, it needs to be initiated with method `init()` and then started with `start()`. This will create a Thread Pool that will make it possible for several connections to be received at the same time. Whenever it is required to stop the Thread Pool, and therefore the processing of new commands, method `stop()` should be invoked on this StreamNodeControInterface instance.

Receiving a command

This Stream Node Control Interface only supports Pipe connections, but it is able to receive a command that is split into several Messages through reordering and joining the content of those Messages.

Eventually a command will be obtained as a byte array that will be passed to the Command Parser. The result of this entity will then be passed to the Sandbox that will take the suitable measures.

Finally, either if the command has failed for any reason or if it has been correctly processed, a Status Message will be sent back through the corresponding Pipe. This Status Message includes three fields:

code or Feedback Code is an int value that describes the outcome of the issued command. This field value will be 0 when the command was successfully executed or a positive integer when it was not.

feedback or Feedback Message is a human readable message regarding the success or failure of the command.

SNPI or Stream Node ProcessLine ID is the Process Line ID coded as a long integer of the actual ProcessLine that was created, stopped or modified by the command. In case of a failed setup command that could not create a new ProcessLine this value will be -1.

5.4.3 Command Parser

For realize the parsing of the commands, [10] proposes use the one that uses the AWML to make the serialization and deserialization of the objects in the Augmented World Model. Instead of the reusage of this component, a new simple method for parsing the commands is implemented. This is because the complexity of AWML, its size and the computational power required. The Figure 5.11 shows the different classes of the new Command Parser.

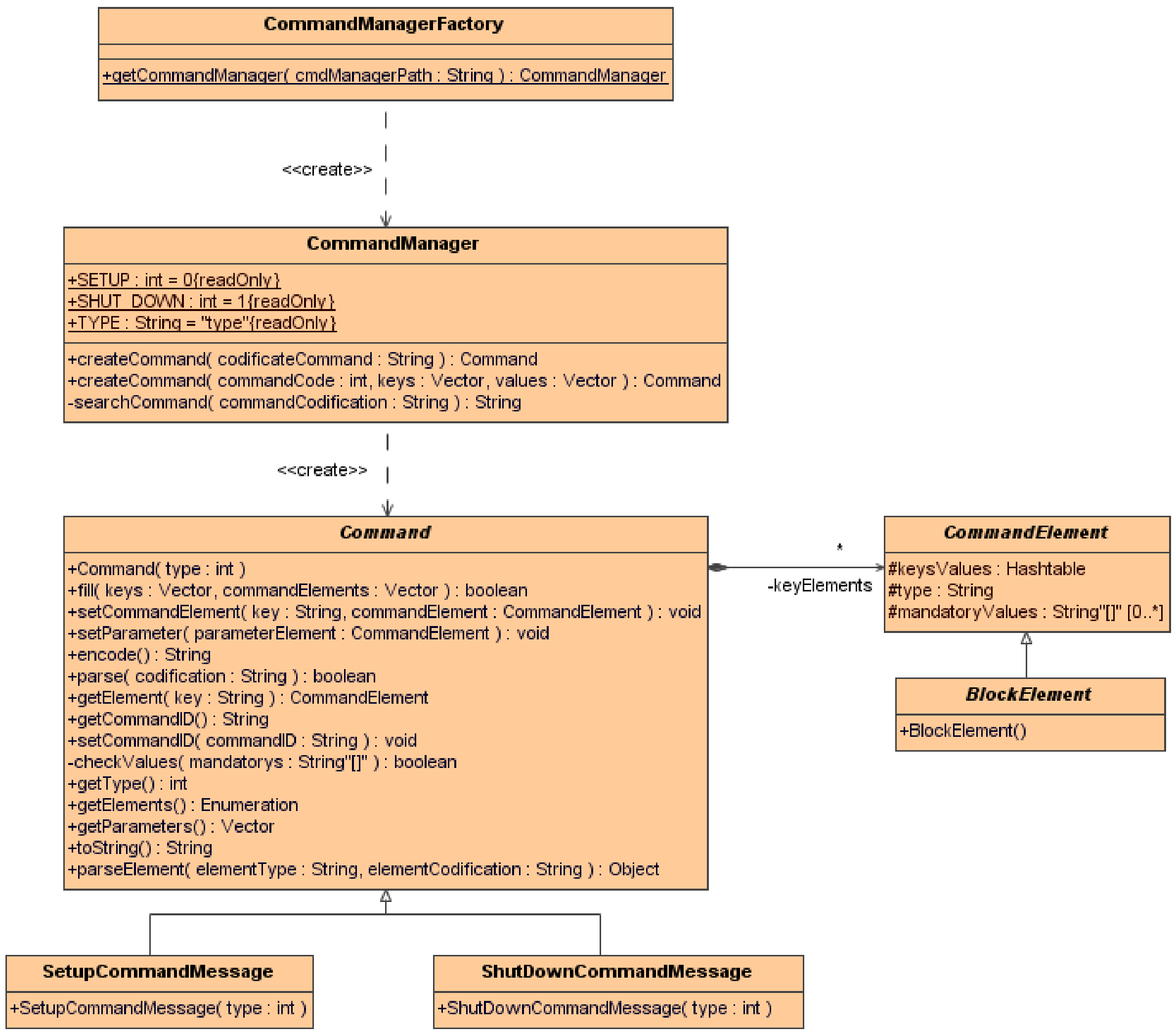

CommandManagerFactory
+getCommandManager(cmdManagerPath : String) : CommandManager
<<create>>
CommandManager
+SETUP : int = 0{readOnly}
+SHUT_DOWN : int = 1{readOnly}
+TYPE : String = "type"{readOnly}
+createCommand(codificateCommand : String) : Command
+createCommand(commandCode : int, keys : Vector, values : Vector) : Command
-searchCommand(commandCodification : String) : String
<<create>>
Command
+Command(type : int)
+fill(keys : Vector, commandElements : Vector) : boolean
+setCommandElement(key : String, commandElement : CommandElement) : void
+setParameter(parameterElement : CommandElement) : void
+encode() : String
+parse(codification : String) : boolean
+getElement(key : String) : CommandElement
+getCommandID() : String
+setCommandID(commandID : String) : void
-checkValues(mandatorys : String"[]") : boolean
+getType() : int
+getElements() : Enumeration
+getParameters() : Vector
+toString() : String
+parseElement(elementType : String, elementCodification : String) : Object
*
-keyElements
CommandElement
#keysValues : Hashtable
#type : String
#mandatoryValues : String"[]" [0..*]
BlockElement
+BlockElement()
SetupCommandMessage
+SetupCommandMessage(type : int)
ShutDownCommandMessage
+ShutDownCommandMessage(type : int)

Figure 5.11: UML class diagram with the main classes of the new Command Parser and the Commands.

Command class : Every concrete command must extend this class. This class defines the structure and behaviour of a command object. A command object is composed by command elements, parameters, a type and a command identifier. In order to define a new specific command class, it is only necessary to extend this class and define by a array of String what kind of elements are mandatory for this new command.

CommandElement class : Every concrete element must extend hits class. Similar to the Command class, this class defines the structure and behaviour of an element object. A element object contains a map with every value with its key, the element type and an array of Strings defining the mandatory values. To create a new concrete Element, it is only necessary extend this class, define the concrete element mandatory values and specify its type.

CommandManager class : This class defines the behaviour of every concrete command manager. Among others,it defines methods to create commands from String with a coded command or from a vector of keys and a vector of values. It also defines a method to identify the type of a command.

CommandManagerFactory class : This class is used to instantiate a concrete CommandManager using a path to the class.

At the present time, only the setup and setdown command are available and working, but defining the update command all new commands is not a complex task as was described above. A setup command has the following parts:

- The description of at least one Data Sink, and at least one Data Source.
- The description of as many Operations as required (can be none), along with their assigned Input Managers.
- The description of the Links that connects the previously described Operations, Data Sinks and Data Sources. This description can indicate the particular type of Blocking Queue needed for this link.
- The description of the Parameters that will configure the Operations, Data Sinks and Data Sources.

In the Appendix A there is an example of Setup Command and Setdown Command with the new format.

5.4.4 Stream Node Sandbox

The Stream Node Sandbox explained in 4.5.3 is modeled with class Sandbox, following the singletone code pattern [8]. This means that only one instance of class Sandbox can exist in the same running Java Virtual Machine. Therefore, in the potential case of having several StreamNodeControlInterface instances running in different Network Groups, all of them will be using the same Sandbox instance.

The Sandbox is prepared to store all the ProcessLine instances that have been created on the StreamNode with the help of a special type of a Map data structure that is able to deal with concurrency issues. But at the moment, as was explained in section 4.5.3, the prototype only permits one process line. This Map is an object that maps keys to values. In this case, the values will be the instances of ProcessLine, and the keys will be the Process Line IDs encoded as long integers. This will allow for setdown or update commands to identify the ProcessLine they refer to

The commands parsed as a Command Object will be processed when the StreamNodeControlInterface invokes method `processCommand()` on the Sandbox instance. This method's first action is to recognize the type of the command issued.

1. In case of a setdown command, the Sandbox stops the desired Process-Line.
2. If the received command is of a modified type, it is passed to the Process-Line.
3. If the command type is a setup, a new ProcessLine is tried to be created by passing its constructor the Command Object, and if it is successful, it will be stored in the internal Map of the Sandbox with a new Process Line ID.

5.4.5 Process Line

The Process Line explained in 4.5.3 is coded in class ProcessLine that indirectly implements the Daemon interface. This makes the ProcessLine an active component, with methods to be initialized, started and stopped, as the StreamNodeControlInterface.

The only constructor method of the class ProcessLine takes a Command Setup as an argument. The first action taken from this constructor will be to load the Boxes, Data Sinks and Data Sources from the information of the Command object. After this step, the Links joining these components will be obtained, and eventually the parameters to configure the Operations, Data Sinks and Data Sources. If any of these steps fail the ProcessLine will not be instanced, and an Exception will be thrown.

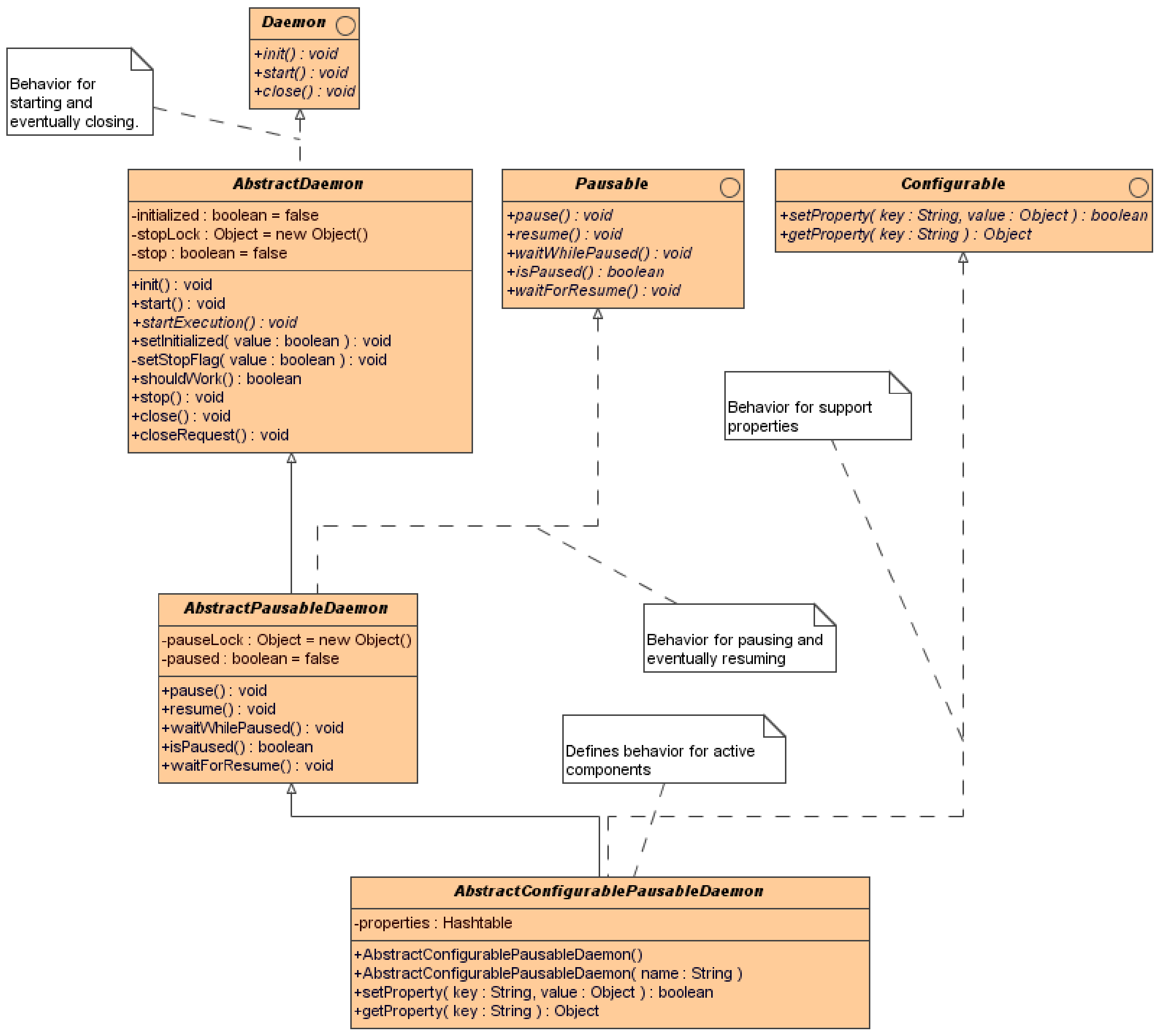

Daemon
+init() : void
+start() : void
+close() : void
Behavior for starting and eventually closing.
AbstractDaemon
-initialized : boolean = false
-stopLock : Object = new Object()
-stop : boolean = false
+init() : void
+start() : void
+startExecution() : void
+setInitialized(value : boolean) : void
-setStopFlag(value : boolean) : void
+shouldWork() : boolean
+stop() : void
+close() : void
+closeRequest() : void
Pausable
+pause() : void
+resume() : void
+waitWhilePaused() : void
+isPaused() : boolean
+waitForResume() : void
Configurable
+setProperty(key : String, value : Object) : boolean
+getProperty(key : String) : Object
Behavior for support properties
AbstractPausableDaemon
-pauseLock : Object = new Object()
-paused : boolean = false
+pause() : void
+resume() : void
+waitWhilePaused() : void
+isPaused() : boolean
+waitForResume() : void
Behavior for pausing and eventually resuming
Defines behavior for active components
AbstractConfigurablePausableDaemon
-properties : Hashtable
+AbstractConfigurablePausableDaemon()
+AbstractConfigurablePausableDaemon(name : String)
+setProperty(key : String, value : Object) : boolean
+getProperty(key : String) : Object

Figure 5.12: UML class diagram of the *AbstractConfigurablePausableDaemon* and the interfaces and classes that extends.

A ProcessLine has the next HastTables to store and organize all the components loaded during the configuration of the process line:

blocks : stores the internal Boxes, Data Sinks and Data Sources having their given name as the key on this mapping, whose three component types will be generally called Blocks. These Blocks implement the Daemon interface, so they are active and then the ProcessLine is being initialized, started or stopped, it will invoke the corresponding methods in each one of the Blocks stored.

inputManagers : stores the Input Managers having their given name as the key on this mapping.

queues : stores the Input Managers having their given name as the key on this mapping.

configurables : stores all the configurable entities such as Sources, Sinks, Operations, Queues and Input Managers.

Data Sources and Sinks

These entities are both modeled by interfaces Source and Sink respectively. Whenever a new type of Data Source or Data Sink needs to be created, the programmer does not need to implement the Source or Sink interfaces, but can use the abstract classes AbstractSimpleSource and AbstractSimpleSink respectively. These abstract classes will be sufficient for creating a single threaded Data Source or a very basic Data Source and will simplify enormously the creation of an implementation of DataSource or DataSink respectively. Figure 5.13 shows the relationship among all this classes.. In the prototype there are two examples of Source and Sink implementations extending these two abstractions.

AbstractPlatformSource This abstraction defines the methods `setNetworkGroup()` and `setLocator()`. At was stated in the design process, for Platform Sources it is mandatory to specify the group where the node is joined and the Connection ID of the remote connection through which the transmission of data will be carried out. This exchange of data is done with Socket connection.

PlatformImageSource This class extends AbstractPlatformSource and implements the method `deserialize()`. In this case, there is not a real deserealization, only returnes the same bytes that receives.

FileOuputStreamSink Extends AbstractSimpleSink and this objective is store the images in the file system. Defines and implements the method `setLocator()` that in this case is used to assign the path of the file where the data will be stored.

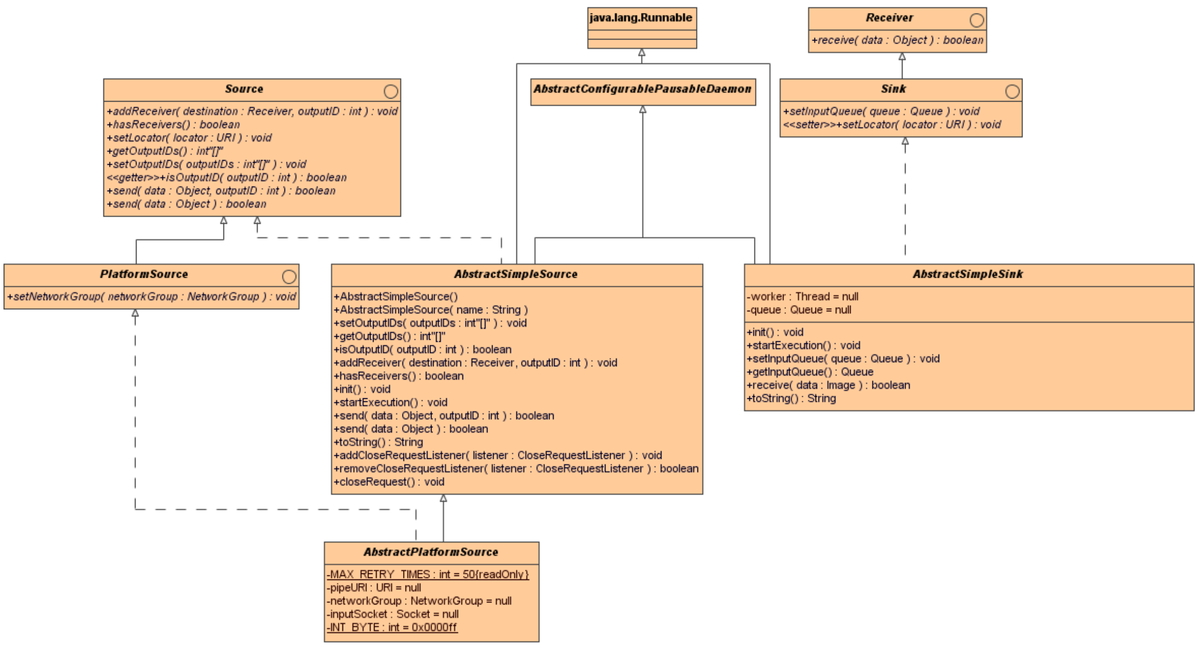

java.lang.Runnable
Receiver
+receive(data : Object) : boolean
Source
+addReceiver(destination : Receiver, outputID : int) : void
+hasReceivers() : boolean
+setLocator(locator : URI) : void
+getOutputIDs() : int"[]"
+setOutputIDs(outputIDs : int"[]") : void
<<getter>>+isOutputID(outputID : int) : boolean
+send(data : Object, outputID : int) : boolean
+send(data : Object) : boolean
AbstractConfigurablePausableDaemon
Sink
+setInputQueue(queue : Queue) : void
<<setter>>+setLocator(locator : URI) : void
PlatformSource
+setNetworkGroup(networkGroup : NetworkGroup) : void
AbstractSimpleSource
+AbstractSimpleSource()
+AbstractSimpleSource(name : String)
+setOutputIDs(outputIDs : int"[]") : void
+getOutputIDs() : int"[]"
+isOutputID(outputID : int) : boolean
+addReceiver(destination : Receiver, outputID : int) : void
+hasReceivers() : boolean
+init() : void
+startExecution() : void
+send(data : Object, outputID : int) : boolean
+send(data : Object) : boolean
+toString() : String
+addCloseRequestListener(listener : CloseRequestListener) : void
+removeCloseRequestListener(listener : CloseRequestListener) : boolean
+closeRequest() : void
AbstractSimpleSink
-worker : Thread = null
-queue : Queue = null
+init() : void
+startExecution() : void
+setInputQueue(queue : Queue) : void
+getInputQueue() : Queue
+receive(data : Image) : boolean
+toString() : String
AbstractPlatformSource
-MAX_RETRY_TIMES : int = 50{readOnly}
-pipeURI : URI = null
-networkGroup : NetworkGroup = null
-inputSocket : Socket = null
-INT_BYTE : int = 0x0000ff

Figure 5.13: UML class diagram of *AbstractSimpleSource*, *AbstractSimpleSink* and *AbstractPlatformSource*.

Operations

The Operation explained in 4.5.3 is the central component in the Stream Node as it is responsible of making the actual processing of the streams of data. Operation interface was created. This interface allows these passive processing components to provide multiple inputs and outputs, as well as a mechanism to receive parameters that can alter the Operation processing, even at execution time.

As the Operation is passive, it does not run in its own thread of execution or implement the Daemon interface. It needs an active entity to invoke its methods. This entity , the Input Manager is the only one calling the Operation methods when it considers appropriate.

In some kinds of Operations the inputs need to be synchronized: no processing is possible until some data from some inputs are ready. A very simple example will be an arithmetic operation that divides two numbers, picking each one from a different input. The Operation interface is prepared to support two different synchronization approaches:

1. The synchronization is done in the Operation: The method `receive()` is invoked by the Input Manager and the method `receive()` calls the method `process()` when all the data necessary from the inputs are ready.

2. The synchronization is done in the Input Manager: The method `process()` is directly invoked by the Input Manager when the data necessary is ready in the inputs.

Whenever a new Operation wants to be coded, the programmer can either implement the Operation interface or extend the abstract class AbstractOperation. This second approach is recommended for its simplicity. In the appendix A there is an implementation of an operation that receives images as a row of bytes and show them consecutively as soon as each one is received. Figure 5.14 shows the UML inheritance diagram of this Operation.

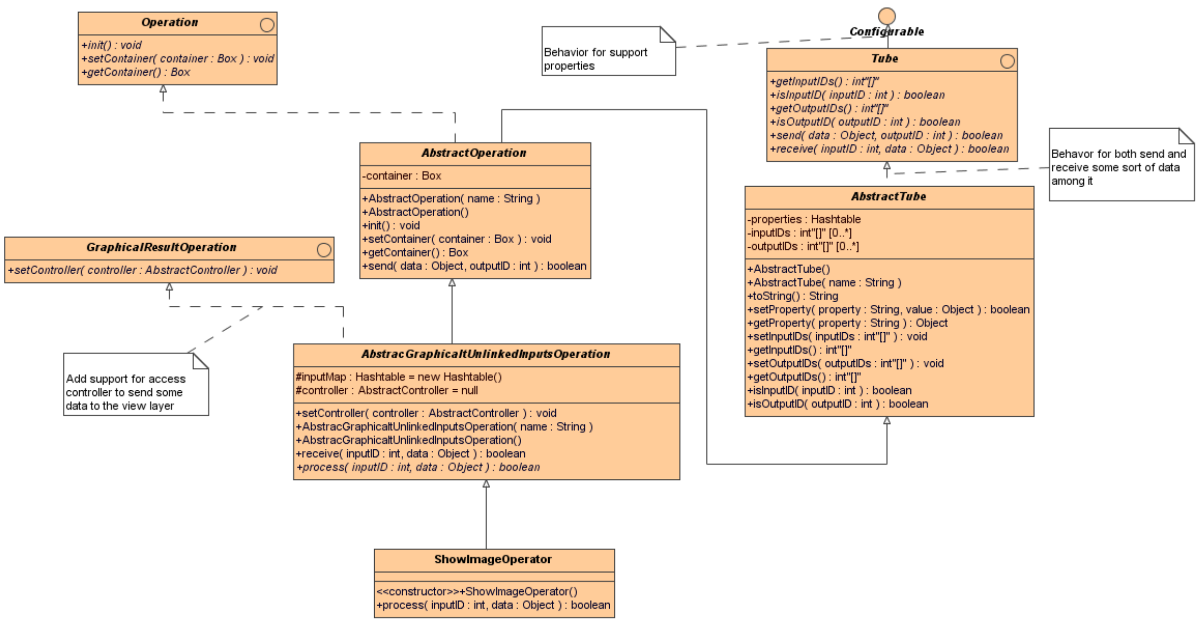

Operation
+init() : void
+setContainer(container : Box) : void
+getContainer() : Box
Behavior for support properties
Configurable
Tube
+getInputIDs() : int"[]"
+isInputID(inputID : int) : boolean
+getOutputIDs() : int"[]"
+isOutputID(outputID : int) : boolean
+send(data : Object, outputID : int) : boolean
+receive(inputID : int, data : Object) : boolean
Behavor for both send and receive some sort of data among it
AbstractOperation
-container : Box
+AbstractOperation(name : String)
+AbstractOperation()
+init() : void
+setContainer(container : Box) : void
+getContainer() : Box
+send(data : Object, outputID : int) : boolean
AbstractTube
-properties : Hashtable
-inputIDs : int"[]" [0..*]
-outputIDs : int"[]" [0..*]
+AbstractTube()
+AbstractTube(name : String)
+toString() : String
+setProperty(property : String, value : Object) : boolean
+getProperty(property : String) : Object
+setInputIDs(inputIDs : int"[]") : void
+getInputIDs() : int"[]"
+setOutputIDs(outputIDs : int"[]") : void
+getOutputIDs() : int"[]"
+isInputID(inputID : int) : boolean
+isOutputID(outputID : int) : boolean
GraphicalResultOperation
+setController(controller : AbstractController) : void
Add support for access controller to send some data to the view layer
AbstracGraphicaltUnlinkedInputsOperation
#inputMap : Hashtable = new Hashtable()
#controller : AbstractController = null
+setController(controller : AbstractController) : void
+AbstracGraphicaltUnlinkedInputsOperation(name : String)
+AbstracGraphicaltUnlinkedInputsOperation()
+receive(inputID : int, data : Object) : boolean
+process(inputID : int, data : Object) : boolean
ShowImageOperator
<<constructor>>+ShowImageOperator()
+process(inputID : int, data : Object) : boolean

Figure 5.14: UML class diagram of *ShowImageOperator*.

The Operation interface extends Configurable because some Operations may require the usage of some configuration Parameters. Most of these Parameters are needed after the Operation is created but before it is initialized with method `init()`. In fact, this initialization process in the Operation is used to check that the required Parameters are already set, and to initialize those data structures that need them. An Operation instance will not be able to process data until it has been initialized, and therefore it has received the required Parameters. But this does not mean that these Parameters will remain unchanged after the initialization process. It should be possible for all Operations to allow the host Box modify those Parameters at any time.

An Operation can receive Parameters through method `setParameter()`. This method accepts parameters from any type, but only String coded Parameters are permitted at the moment because of the limitations of the String based commands that the Stream Node accepts.

Box

The Boxes described in 4.5.3 are represented by an interface called Box. Box implements the Daemon interface and is considered as an active component. The prototype has only one implementation of this interface, the QueuedBox. This component is not described by a setup command as it is created and used internally by a ProcessLine.

The QueuedBox can host one InputManager, one Operation and as many BlockingQueues as inputs have the Box. As the InputManager is an active component, it should be initialized, started and stopped.

When the process line initialites the Box by the method `init()`, the next actions are done by the Box which:

- Checks that the Operation is created and initializes it like the active component it is.
- Checks that the InputManager is created and initializes it like the active component it is.
- Checks that there is one BlockingQueue for each input

Whenever a Box is asked to be initialized, started or stopped, it will invoke the respective method of the InputManager.

This element is responsible of handling the connections between the internal Operation and other Boxes, Data Sources and Data Sinks. In order to allow these connections, several Receivers can be registered for each one of the outputs of the box. This resembles the observer code pattern described in [8]. As soon as some data element wants to be sent through a defined output, all the Receivers associated to this output will be notified, receiving this way the data element. The interface Receiver is implemented by every type of Sink, and by the class BoxReceiver. This last class will act as a wrapper of a Box and one of its inputs, so each object that is received by this wrapper will be passed to the defined input of the wrapped Box. Whenever a data element is received in one input, the Box will put it in the corresponding BlockingQueue. The relationship between all those classes, together with the rest of the internal components of a Process Line can be seen in Figure 5.16.

Input Managers

The idea of the Input Manager exposed in 4.5.3 is represented by interface InputManager. As it was already mentioned, the Input Manager is the active component inside a Box that is responsible of taking data elements from the Blocking Queues and passing them as an argument to the `receive()` method of the Operation they are associated with.

As shown in Figure 5.15, the class AbstractInputManager extends indirectly the Daemon interface, so it will provide methods to be initialized, started and stopped. If an InputManager makes use of Java Thread objects, these methods will be respectively used to create, start and destroy all the contained Threads.

The default Input Manager is represented by class *ParallelAsyncInputManager* whose UML inheritance diagram is shown in Figure 5.15. This Input Manager implementation uses its internal class WorkerThread to do the work.

The class WorkerThread extends the Java class Thread and as many WorkerThreads are created as inputs have the Box. The current implementation is prepared to have several WorkerThread but at the moment, only one WorkerThread is permitted to the single input that a box can have. These limitations are included in the restrictions imposed due to the capacities of the mobile devices.

The behaviour of each WorkerThread running in the InputManager is:

1. Each time a queue is not empty, the worker gets an element from the queue.
2. It invokes the `receive()` method of the Operation instance, passing as arguments of this method both the input number and the data element.
3. It remains locked till the Operation finishes the processing of the data element.
4. When the method returns, the worker tries again to obtain another data element from the queue and so on.

The DefaultInputManager creates as many WorkerThread instances as inputs have the Box.

The DefaultInputManager can be used with any Operation that needs no input synchronization or handles this synchronization internally. This class can be used as a template to create an InputManager that will handle the synchronization of the inputs for an specific Operation

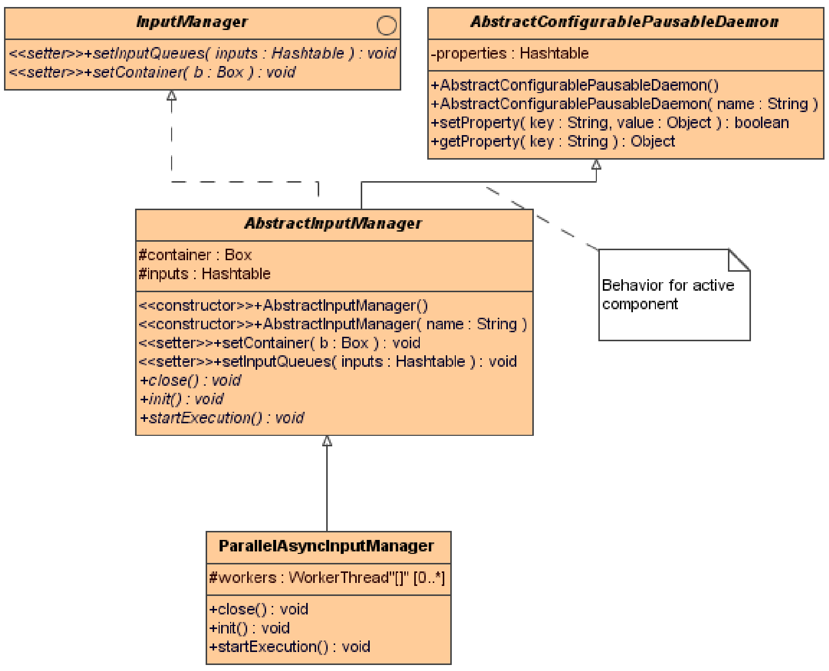

InputManager
<<setter>>+setInputQueues(inputs : Hashtable) : void
<<setter>>+setContainer(b : Box) : void
AbstractConfigurablePausableDaemon
-properties : Hashtable
+AbstractConfigurablePausableDaemon()
+AbstractConfigurablePausableDaemon(name : String)
+setProperty(key : String, value : Object) : boolean
+getProperty(key : String) : Object
AbstractInputManager
#container : Box
#inputs : Hashtable
<<constructor>>+AbstractInputManager()
<<constructor>>+AbstractInputManager(name : String)
<<setter>>+setContainer(b : Box) : void
<<setter>>+setInputQueues(inputs : Hashtable) : void
+close() : void
+init() : void
+startExecution() : void
Behavior for active component
ParallelAsyncInputManager
#workers : WorkerThread"[]" [0..*]
+close() : void
+init() : void
+startExecution() : void

Figure 5.15: UML class diagram of *ParallelAsyncInputManager*.

Blocking Queues

The idea of the blocking queues was taken from the Java interface BlockingQueue that is explained in [18]. The queues that implement this interface must deal with synchronization problems and provide some blocking methods to obtain or add elements to the queue.

Java Micro Edition does not implement blocking queues, therefore its own implementation has been created. For that purpose a Queue interface was created defining the methods offer, poll, put, take and setCapacity. An implementation of this class and the default blocking queue in the prototpye is the StreamBlockingQueue class. The behaviour follows the criteria of the BlockinQueue of Java Standart Edition API.

The method `take()` takes an element from the queue, and if the queue is empty, it will wait till a new element is received. This method is interesting as it allows a thread to block just the needed time, avoiding busy waiting techniques that will invoke the `remove()` method repeatedly till some data is obtained.

In addition, the method `offer()` allows to specify a maximum waiting time if necessary for space to become available on the queue. Similarly, method `put()` blocks for an unlimited amount of time till some space is available on the queue. These methods can be useful in such applications where it is not desired for elements to be dropped.

The StreamBlockingQueue has two different constructors. One permits specifying the maximum number of elements allowed and the second one establishes the max value of an Integer. Some elements can be configured with parameters, and in this case, a setup command can contain the size parameter for each Blocking Queue used.

Receiver

The Receiver interface is an abstraction used to allow a Box to send data through one of its output without taking into account if the receiver component is a Sink or a Box. In the prototype only the interface Sink and the class BoxReceiver extends the Receiver class.

The main difference between these Receiver wrappers will be the way to put data into the Blocking Queue that corresponds to this Receiver. If a lossless connection is required, then method `put()` of class BlockingQueue is the best choice. If the connection is at loss, meaning that dropping elements will not affect the performance of the processing, the method to be used will be `offered()` without specifying a timeout. If a connection should not loose too many elements, the most appropiate method to use will also be `offered()`, but in this case specifying a maximum waiting time.

But the BoxReceiver cannot access directly to a BlockingQueue, as they are wrapped within the Box. Because of this, in order to allow different Receiver implementations for loss and lossless links, it will be needed to modify the interface Box to provide different receive methods.

5.5 Stream Federation MIDlet

The next points explain the implementation of the MIDlet, which is, as was stated in the point 2.2.5 the application that runs in the mobile device. At this point, we can differentiate between the implementation destined to the user interface and the classes where the Stream Platform and the Stream Node Pool are. Therefore, the MIDlet at a whole is designed following the architecture pattern called Model View Controller (MVC). This pattern divides the system in three layers and applied to our case will be:

View layer This layer includes all the classes assigned to the graphical interface offered to the user through the display of the mobile device.

Controller layer The classes that are defined here are those which will communicate the view layer and the model layer. In some cases, this layer receives some data from the user that must be processed by the model layer, and in other cases, the model layer will send some data to notify changes to the user.

Model layer Also called business layer, will be composed by the Stream Platform and the Stream Node Pool implementation.

In Figure 5.16 we can see the relation between the layers and the classes belonging to them. The next points explain more in depth some of the classes of the view layer and the controller layer.

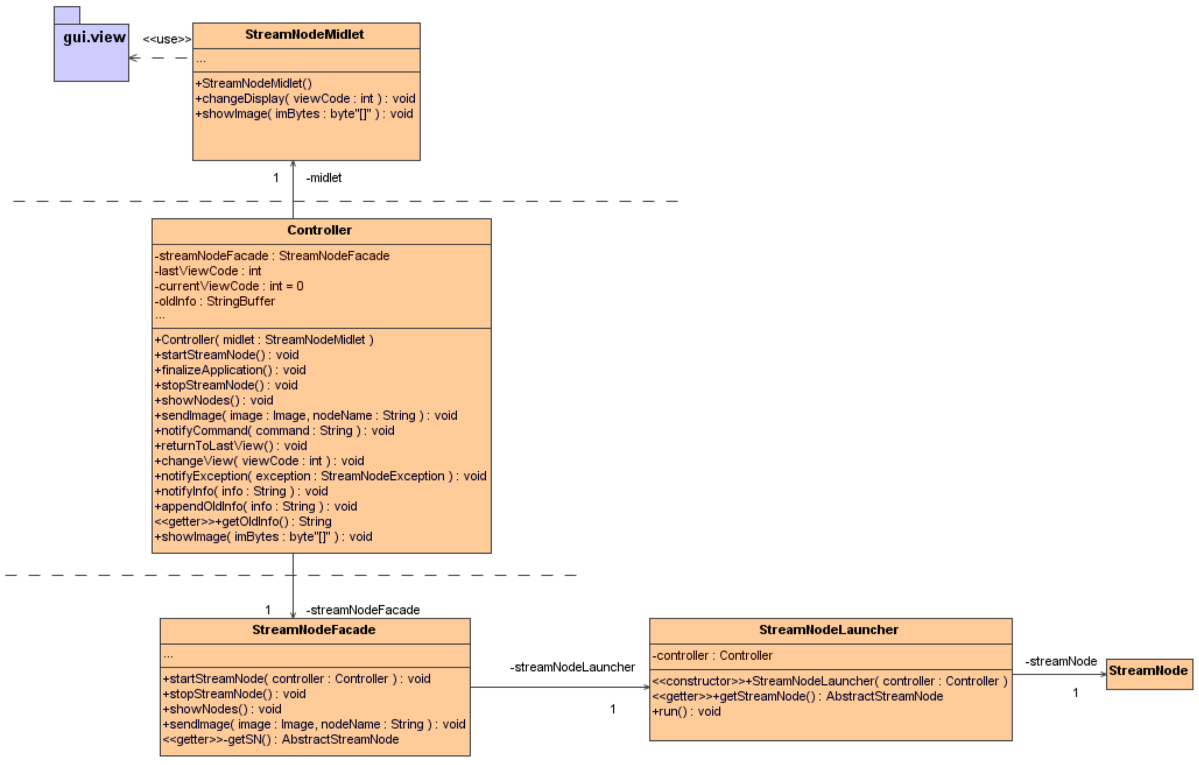

gui.view
<<use>>
StreamNodeMidlet
...
+StreamNodeMidlet()
+changeDisplay(viewCode : int) : void
+showImage(imBytes : byte"[]") : void
1
-midlet
Controller
-streamNodeFacade : StreamNodeFacade
-lastViewCode : int
-currentViewCode : int = 0
-oldInfo : StringBuffer
...
+Controller(midlet : StreamNodeMidlet)
+startStreamNode() : void
+finalizeApplication() : void
+stopStreamNode() : void
+showNodes() : void
+sendImage(image : Image, nodeName : String) : void
+notifyCommand(command : String) : void
+returnToLastView() : void
+changeView(viewCode : int) : void
+notifyException(exception : StreamNodeException) : void
+notifyInfo(info : String) : void
+appendOldInfo(info : String) : void
<<getter>>+getOldInfo() : String
+showImage(imBytes : byte"[]") : void
1
-streamNodeFacade
StreamNodeFacade
...
+startStreamNode(controller : Controller) : void
+stopStreamNode() : void
+showNodes() : void
+sendImage(image : Image, nodeName : String) : void
<<getter>>-getSN() : AbstractStreamNode
-streamNodeLauncher
1
StreamNodeLauncher
-controller : Controller
<<constructor>>+StreamNodeLauncher(controller : Controller)
<<getter>>+getStreamNode() : AbstractStreamNode
+run() : void
-streamNode
1
StreamNode

Figure 5.16: UML class diagram with the main classes of the Model, View and Controller layers of the *MIDlet*.

5.5.1 StreamNodeFacade and StreamNodeLauncher

Both classes are belonging to the model layer and provide methods for accessing to the Stream Node.

The StreamNodeFacade is an interface that defines a set of operations offered to the controller in order to interact with the Stream Node. The StreamNodeFacade delegates its operations to the StreamNodeLauncher that is the class with access to the StreamNode.

The StreamNodeLauncher is class responsible of launching the StreamNode as a new thread. This is because the MIDlet cannot depend of an operation of input or output in order to avoid blocking problems. These possible input and output are made, for example, for reading and saving the configuration of the Stream Node and the Stream Platform.

5.5.2 Controller

Both view layer classes and model classes will communicate with this class in order to interact between them. The Controller has a reference to the StreamNodeFacade and another reference to the StreamNodeMidlet. The figure reffig:view shows this class and its methods.

Another function of this class is to store some state information, like the current view that is activated in the MIDlet, the last view that was active in the MIDlet and a StringBuffer where information that was passed to a view and was not possible to show is stored. This buffer allows to show the information in the correct moment and in the proper view.

5.5.3 MIDlet and views

As was explained in the point 2.2.5, there must be at least one class that extends the class javax.microedition.midlet.MIDlet of the J2ME API. In our prototype this class is the AbstractMidlet, which also defines some behaviour. This abstract class is implemented by the StreamNodeMidlet class.

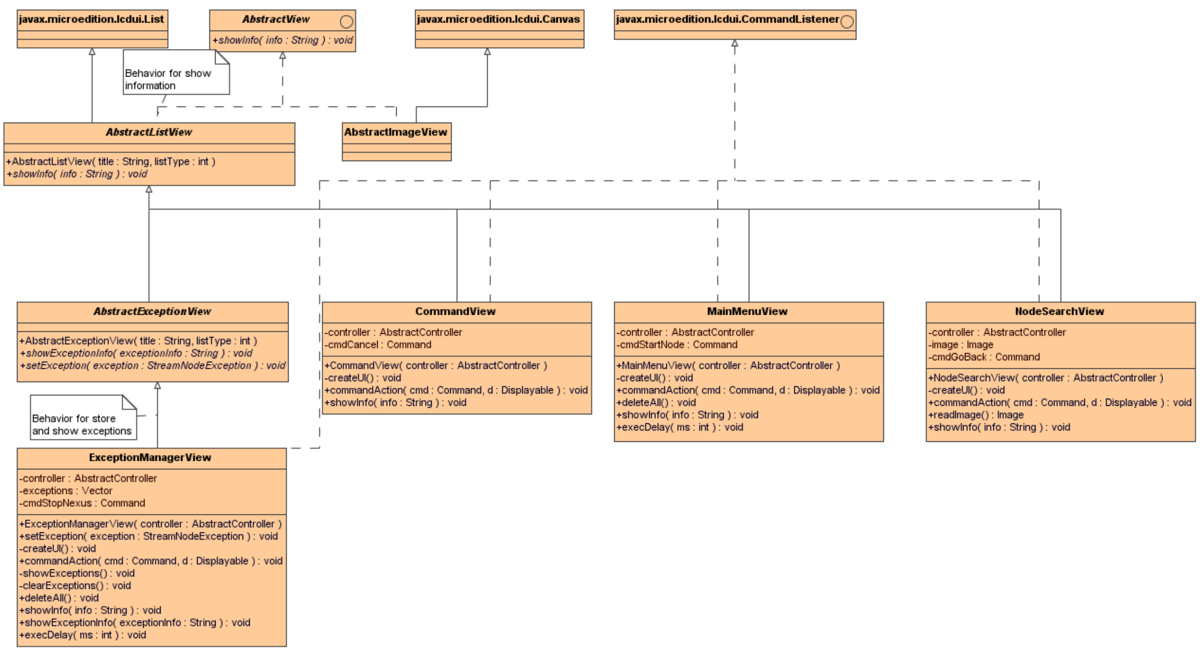

javax.microedition.lcdui.List
AbstractView
+showInfo(info : String) : void
javax.microedition.lcdui.Canvas
javax.microedition.lcdui.CommandListener
Behavior for show information
AbstractListView
+AbstractListView(title : String, listType : int)
+showInfo(info : String) : void
AbstractImageView
AbstractExceptionView
+AbstractExceptionView(title : String, listType : int)
+showExceptionInfo(exceptionInfo : String) : void
+setException(exception : StreamNodeException) : void
Behavior for store and show exceptions
ExceptionManagerView
-controller : AbstractController
-exceptions : Vector
-cmdStopNexus : Command
+ExceptionManagerView(controller : AbstractController)
+setException(exception : StreamNodeException) : void
-createUI() : void
+commandAction(cmd : Command, d : Displayable) : void
-showExceptions() : void
-clearExceptions() : void
+deleteAll() : void
+showInfo(info : String) : void
+showExceptionInfo(exceptionInfo : String) : void
+execDelay(ms : int) : void
CommandView
-controller : AbstractController
-cmdCancel : Command
+CommandView(controller : AbstractController)
-createUI() : void
+commandAction(cmd : Command, d : Displayable) : void
+showInfo(info : String) : void
MainMenuView
-controller : AbstractController
-cmdStartNode : Command
+MainMenuView(controller : AbstractController)
-createUI() : void
+commandAction(cmd : Command, d : Displayable) : void
+deleteAll() : void
+showInfo(info : String) : void
+execDelay(ms : int) : void
NodeSearchView
-controller : AbstractController
-image : Image
-cmdGoBack : Command
+NodeSearchView(controller : AbstractController)
-createUI() : void
+commandAction(cmd : Command, d : Displayable) : void
+readImage() : Image
+showInfo(info : String) : void

Figure 5.17: UML class diagram of view interfaces and some specific implementations.

In the packet gui there are several abstract classes that must be extended by all the views that we want to include in the MIDlet. One common operation of the views is to show information in the display. But not every kind of view represents the information in the same way or it is simply not possible to show it. Because of this, some abstract classes are defined that determine some common methods and then there are some view implementations of these abstractions in order to correctly define the methods with different behaviour. The figure 5.17 shows this hierarchic implementation and an example of different implementations used in the prototype.

Finally, the class StreamNodeException wraps information about an error during the execution, like the reason of the error, where it ocurred and the concrete information of the possible java exception launched. This class is very useful in order to be able to know what happened internally because all this information is presented in the display of the device. This error information is filled in the model and sent to the view through the controller.

5.5.4 Actions

These classes are implemented with the objective of being able to send commmands and images to other StreamNode in order to test the prototype. The structure and behaviour is not very complex. First, there are some actions that are designed to carry out different simple tasks.

- ReadImageAction: Reads images from the file system of the device.
- SendImageAction: Receives the images (as bytes) from ReadImageAction and sends them through a reliable socket connection.
- SendCommandAction: Defines how to send a command. SendSetupCommandAction and SendSetDown-Command extend this class and only needs to implement the method `createCommand()` in order to create the specify command.

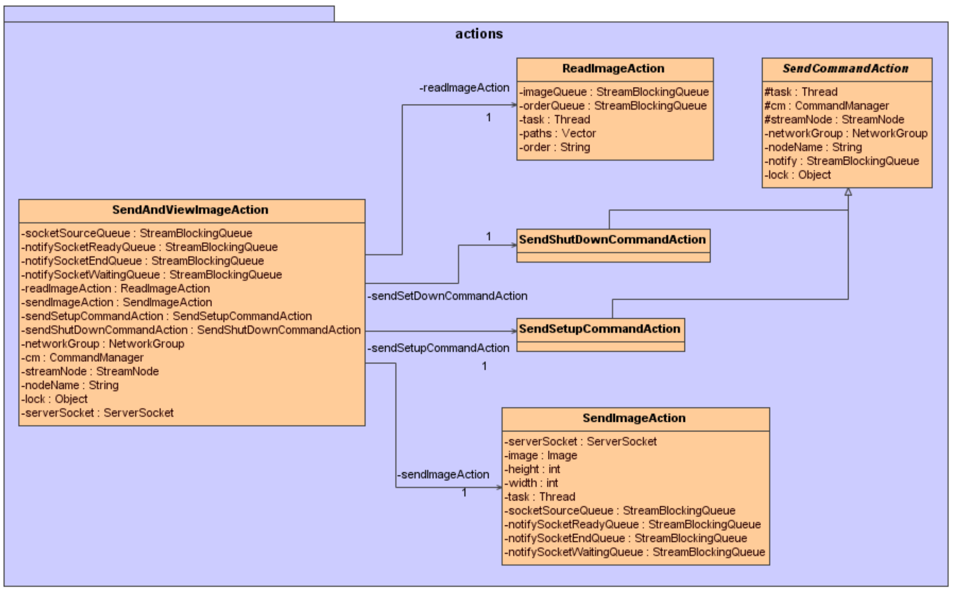

actions
ReadImageAction
-imageQueue : StreamBlockingQueue
-orderQueue : StreamBlockingQueue
-task : Thread
-paths : Vector
-order : String
SendCommandAction
#task : Thread
#cm : CommandManager
#streamNode : StreamNode
-networkGroup : NetworkGroup
-nodeName : String
-notify : StreamBlockingQueue
-lock : Object
-readImageAction
1
SendAndViewImageAction
-socketSourceQueue : StreamBlockingQueue
-notifySocketReadyQueue : StreamBlockingQueue
-notifySocketEndQueue : StreamBlockingQueue
-notifySocketWaitingQueue : StreamBlockingQueue
-readImageAction : ReadImageAction
-sendImageAction : SendImageAction
-sendSetupCommandAction : SendSetupCommandAction
-sendShutDownCommandAction : SendShutDownCommandAction
-networkGroup : NetworkGroup
-cm : CommandManager
-streamNode : StreamNode
-nodeName : String
-lock : Object
-serverSocket : ServerSocket
SendShutDownCommandAction
-sendSetDownCommandAction
SendSetupCommandAction
-sendSetupCommandAction
SendImageAction
-serverSocket : ServerSocket
-image : Image
-height : int
-width : int
-task : Thread
-socketSourceQueue : StreamBlockingQueue
-notifySocketReadyQueue : StreamBlockingQueue
-notifySocketEndQueue : StreamBlockingQueue
-notifySocketWaitingQueue : StreamBlockingQueue
-sendImageAction

Figure 5.18: UML class diagram of *SendAndViewImageAction* and its internal action classes.

Every one of these tasks extends AbstractDaemon class, so it will provide methods to be initialized, started and stopped; furthermore it implements the interface runnable, therefore, they are asynchronous between them. For synchronizing and establishing an order, BlockingQueues are used. For example, to send the images read in ReadImageAction and send them to SendImageAction.

Then, another action wraps the actions before they are defined, instantiating and preparing them and finally starting them. This action is the class SendAndViewImageAction. The main methods are:

- prepareAction(): Initializes all the Blocking Queues, all the actions above described and calls the method `init()` of every one.
- startAction(): Invokes the method `start()` of all the actions previously prepared.

CHAPTER 6

PROTOTYPE EVALUATION

Among other things, at the time of writing this document, a Stream Node is able to:

- Join the network and the net peer group.
- Discover other nodes in the group.
- Send setup commands and setdown commands to another node.
- Receive setup commands and setdown commands from another node.
- Configure and run one process line at a time.

In the following points, what will be explained is how the prototype is tested with a real example of stream processing, where the points above commented can be seen working.

6.1 Work Environment

The next devices are used in order to carry out the evaluation of the prototype.

- Laptop with Intel Core Centrino 2 Duo 7200, 2GHz, 4 MB cache and 2GB of main memory DDR2-667. The OS is Ubuntu 8.10 "the Intrepid Ibex" 64bits, with a kernel 2.6.27-14-generic . The software installed is: Java SE version 1.6.0_10-b33, Java Wireless Toolkit for CLDC version 2.5.2_01 and the JXME proxyless version 2.5.
- HTC MDA with proccesor Intel PXA270 of 520 Mhz, 64 MB of main memory. The OS is Microsoft Windows Mobile Version 5.0 and the application for running the MIDlets is MIDlet HQ.
- Two mobile device emulators (running in the laptop).
- 54 Mbps Wireless Router 802, 11 g, configured with WPA security.

6.2 Scenario and objectives

Four Stream Nodes are launched. The first one, called LaptopNode, is launched in the laptop and is configured like rendezvous. Another node, called MDANode, is launched in the HTC MDA mobile and two more, called ENode1 and ENode2, are launched in the emulators that are running in the laptop. All the devices are connected with a local network using a wireless connection offered by the router. The picture 6.1 represents the situation with all the participants.

The test and evaluation consist in transferring a set of 23 consecutive Meteosat images from Enode1 to MDANode, showing these images in MDANode and storing them in the non-volatile memory of MDANode. The following steps are carried out in order to achieve this stream processing:

1. Launch all the Stream Nodes.
2. Discover nodes in the network with the Enode1.
3. Send setup command from Enode1 to MDANode.

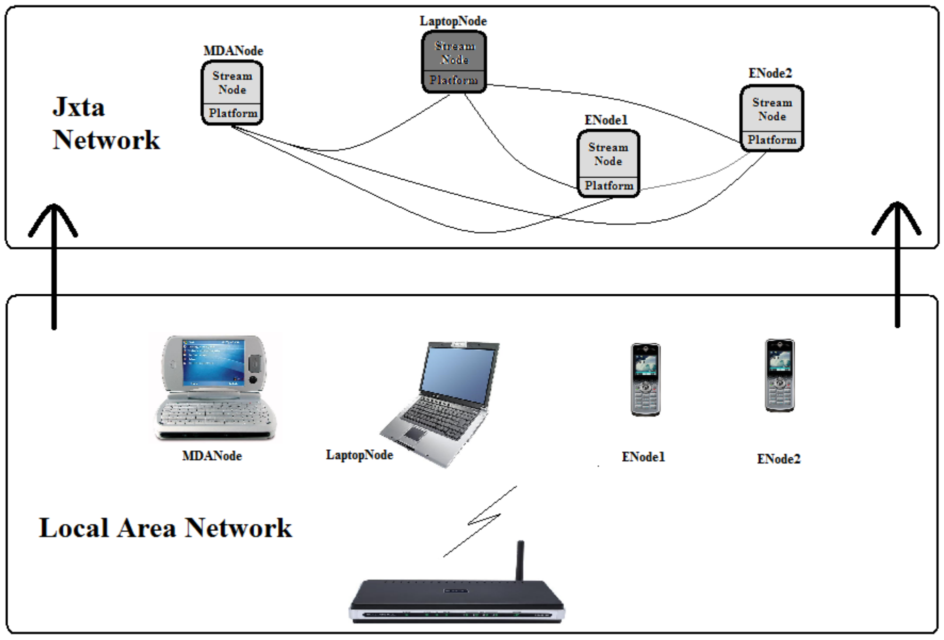


Figure 6.1: Scenario where the protototype is tested.

4. When MDANode is configured, the stream of images starts to be sent from Enode1.
5. Send setdown command from Enode1 to MDANode to finish the Process Line in MDANode.

The commands are sent from Enode1 and MDANode through a Pipe connection. This connection is established when Enode1 discovers MDANode and gets the ConnectionID published before by MDANode in its node information. For transferring the images, a pipe connection is not the best approach, therefore a reliable socket is used. Enode1 creates a server socket that waits for connections. When MDANode configures all the blocks in the process line and this starts running, the Data Source will establish the connection with the server socket.

The next principal elements are defined in the setup command in order to configure the Process Line in MDANode:

Data Source : The data source indicated is a PlatformImageSource. This source will open a client socket to the server socket opened before in Enode1 and receive the stream data.

Data Sink : The data sink indicated is a FileOutputStreamSink. This sink receives the bytes of the images and stores each image in the file system.

Operation : The operation indicated is a ShowImageOperator. This operation takes bytes of the images from the Data Source, then shows each photo in the display, and finally sends the bytes to the Data Sink.

Input Manager : The Input Manager indicated is a ParallelAsyncInputManager. This Input Manager is restricted to manage only one input with only one thread.

Blocking Queue : The Blocking Queue indicated for input and output channels is a Stream Blocking Queue. The size of the queue is also specified to a value of 2.

Links : Two links are indicated. The first is to concrete the connections between the source, queue and operation. The second is to specify the connections between the sink, queue and operation.

CHAPTER 7

CONCLUSIONS

The objective of this thesis has been to develop a framework for mobile data stream processing by analyzing, designing, implementing and testing a functional prototype. At this point, we can say that this primary goal has been achieved.

The prototype implemented can be considered as a Proof of Concept, demonstrating the feasibility of the design carried out and providing a milestone on the way to a fully functioning prototype. Among others, at the present we can create a pool of Stream Nodes, some of them running in mobile devices and others in desktop computers, these last Stream Nodes belonging to the framework described in [10] therefore, another goal has been achieved: the integration in the NEXUS project. With all the nodes interconnected, it is possible to accomplish discovery functions in order to find and identify these nodes and send messages between them. These messages can be setup or setdown commands. With the setup message, a node can be configured to accept stream data from another and carry out a stream processing of this flow of data. Of course, this implementation is very limited, not valid for the real purposes inside the NexusDS project, and future work is necessary in order to improve, refine and test intensively.

Another important work was the research carried out on the JXTA version for mobile devices. It was necessary to make an important work of testing and patching over this platform in order to know if it would accomplish its purposes inside the framework. This investigation proved to be successful, offering the opportunity for other NexusDS developers to use this technology knowing its real capabilities.

7.1 Future work

7.1.1 Wide performance test

Working in mobile devices, such as phones or PDAs, the performance is a delicate and complicated point to take into account. A variety of tests must be done over the framework in order to see the performance that can be achieved in the system. This testing task should be continuously done detecting possible deterioration of performance due to any changes or aggregation of new components.

7.1.2 Achievement and approach for dynamical operations download

As was commented several times during the different development phases, J2ME does not permit download code dynamically and then instantiate the new classes. This capacity that is described in [10] is an interesting idea and could be achieved with some kind of approach, or at least a method could be found to include new operations.

One idea could be include a special service into the NEXUS platform. This service can provide different versions of the MIDlet storing them in something like a “MIDlet distribution repository”. Each version could be defined by the operations that it implements. For example, if a client requires having new operations, he can use this server. Perhaps he must send some information about his MIDlet version in order to compare the current operations and the new ones. Then, the user must update the MIDlet manually. There is no problem with the platform configuration because the files should be stored in a non-volatile memory of the device.

Obviously, this process is neither very dynamic nor quick, but it can offer a way to get new operations or simply new versions.

7.1.3 Measuring time responses

Currently, there are no time response measurements. Probably, these times do not fulfill the requirement of a short time response. Therefore, a set of test must be designed in order to make measurements in tasks like sending commands or large files, or the time involved in configuring a process line.

7.1.4 Continue optimization of code

One requirement defined in the requirements chapter is to establish that the code be optimized in order to get good results and save resources. Along the implementation phase, some of the basic methods to optimize the code were taken into account, for example, reusing objects, closing socket and file connections, collections with the exact size necessary, avoiding loops and reducing the code size. Any way, this is not sufficient and more strategies must be taken into account. And finally, another good practice is to try to analyze the system resources in different points of code and find where there are possible bottle necks or a drastic reduction of resources.

7.1.5 Synchronization review

Due to limitations of J2ME in terms of collections, some code lines are using collections that are not synchronized. This situation can produce problems in the future which can not be seen now. A review should be made, not only of these collections but of other situations where synchronization is necessary, and moreover, where synchronization methods can be avoided in order to achieve better performance.

7.1.6 Alternatives to jsr75 profile

As was explained in point 2.2.4, this profile offers an API to manage files in an easy way in the file system of the device, giving more flexibility to the system. But this API does not work in all devices, limiting the compatible devices and being contrary to the requirement of having a wide range of mobiles compatible with the framework.

Two alternatives could be:

1. An RMS that provides a mechanism through which the MIDlets can persistently store data and retrieve it later, following a record-oriented approach. RMS is not flexible and transparent, and the files are identified by the number of registry. There are some third party APIs in order to use RMS in an easier way, but they do not eliminate some restrictions on file sizes and others.

2. Save online, for example by http, the important files like configuration files.

Both alternatives can only possibly be used with files which are not very large, so the system is not able to store data like photos or video.

7.1.7 Control for leaving and joining nodes

This is not an easy task but must be taken into account because there will be nodes joining and leaving the system. What happens if some subtask is being processed in one node that leaves the platform?. We must have some mechanism to know that this task has not ended and dynamically send it again to other node.

7.1.8 Implement a strategy for fault tolerance

The current implementation does not offer fault tolerance and only offers simple mechanisms of java exceptions to catch errors and notify the user or try to follow another way. So that is not a fault tolerance system and it does not accomplish the requirement described in 3.3.

7.1.9 Optimal data transmission

A research on the optimal size of the chunks sent through sockets is important in order to try to reduce the transmission times in large files. Some aspects can be MTU of the connectivity platform or analyzing how the JxtaSockets work in terms of how they split the data.

7.1.10 Fix some JXME errors

Among other things, it is important to fix the problem by creating and joining new groups over the network. This is important to follow the design of the Stream Federation.

7.1.11 Take advantage of JXME collections

JXME has some collections in its API that MIDP profile or the CLDC configuration does not include. These collections are similar to those in the API of J2SE. Currently they are used only in few points of the code because it was not known if they operated correctly.

7.2 Acknowledgments

I am pleased to again give my thanks to many of those people who have been supporting me during these 6 years of studies, especially my family.

First, I dedicate this thesis to my parents and brother. They always have helped me, supported and guided since I started to study in 2003. I cannot measure all the confidence they placed in my day to day, always providing everything that I needed. I hope I have many occasions to give them thanks again. I do not forget my grandparents, who always remain to me and bring me so much affection.

No less satisfying is able to include here all these new people I have met during this amazing year. They were always there beside me, offering their support in the most difficult moments. I can not write the name of all them, but I would put at least some of those good friends: Carlos Sarti, David Iglesias, Pablo Torné, Jacobo Vazquez, Juan Gavilán, Sergio León, Adrián Llorente, Cristina Prada, Ferran Peláez and Alfonso Sanz

My most sincere gratitude to my supervisors Dipl. Inf. Harald Weinschrott and Dipl. Inf. Nazario Cipriani for giving me the opportunity to realize project at the institute IPVS, consisting of a fantastic group of people that always treated me great during all the last year.

Finally, my thanks to Consuelo Martinez for his contribution so important in the last days.

BIBLIOGRAPHY

[1] Project JXTA JavaTM (2007). *JavaTM Standard Edition v2.5:Programmers Guide*, 2007.

[2] Sun Microsystems (2009). *Java 2 Micro Edition*, 2009.

[3] Ball J. Bodoff S. Carson D. B.-Evans I. Green D. Haase K. Armstrong, E. and E. Jendrock. *The J2EETM 1.4 Tutorial*, electronic version: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/jms.html edition, 2008.

[4] Babu S. Datar M. Motwani R. Babcock, B. and J. Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*.

[5] Durr F. Geiger J. Grossmann M. Honle-N. Joswig J. Nicklas D. Bauer, M. and T. Schwarz. Information management and exchange in the nexus plaform. Technical Report 2004/0. 15, Universitat Stuttgart, 2004.

[6] Fielding R. Berners-Lee, T. and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*, electronic version: http://www.ietf.org/rfc/rfc3986.txt edition, 2005.

[7] Andreas Brodt Matthias Grossmann Bernhard Mitschang Cipriani Nazario, Mike Eissele. *NexusDS: A Flexible and Extensible Middleware for Distributed Stream Processing*. Universitat Stuttgart, 2009.

[8] J. W. Cooper. *The Design Patterns Java Companion*. Addison-Wesley, 1998.

[9] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 1992.

[10] Daniel Garcia Sardina. Framework for distributed data processing. Master's thesis, 2008.

[11] Polk W. Ford W. Housley, R. and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, electronic version: http://www.ietf.org/rfc/rfc3280.txt edition, 2002.

[12] Project JXTA. *JXTA Java Micro Edition*, 2008.

[13] T Nis. *Dictionary of algorithms and data structures*. 2005.

[14] Wetterberg Erik Pleumann Jorg, Yadan Omry. *Antenna An Ant-to-End Soluction For Wireless Java*. Electronic version: http://antenna.sourceforge.net/, 2002-2009.

[15] The JXTA Project. *JXTA v2.0 Protocols Specification*. Electronic version: https://jxta-spec.dev.java.net/JXTAProtocols.pdf, 2007.

[16] 2000 Sun Microsystems. *J2ME Building Blocks for Mobile Devices White Paper on KVM and the Connected*. Sun Microsystems, 2000, 2000.

[17] Ugur Cetintemel Mark Humphrey Jeong-Hyon Hwang Anjali Jhingran Anurag Maskey Olga Papaemmanouil Alex Rasin Nesime Tatbul Wenjuan Xing Ying Xing Stan Zdonik Yanif Ahmad, Bradley Berg. Distributed operation in the borealis stream processing engine (invited demonstration). In *2nd International Conference on Geosensor Networks*.

[18] Hommel S. Royal J. Rabinovitch I. Risser-T. Zakhour, S. and M. Hoeber.

Appendix A

Samples

A.1 Setup command

This section contains an example of a encoded Setup Command and a encoded Setdown Command.

```
&commandType=|commandType=0|&link1=|inputBlockID=testSourceID|inputBlockOutputSlotID=0|
    outputBlockID=testOperationID|outputBlockInputSlotID=0|inputQueueID=testQueueID|&link2=|
    inputBlockID=testOperationID|inputBlockOutputSlotID=0|outputBlockID=testSinkID|
    outputBlockInputSlotID=0|inputQueueID=testQueueID|&source=|blockID=testSourceID|classURI=
    urn:java:de.uni_stuttgart.nexus.streamFederation.sources.basic.platformImage.
    PlatformImageSource|blockType=source|remoteLocator=urn:jxta:uuid-59616261646162614
    E5047205032503393B5C2F6CA7A41FBB0F890173088E79404|&sink=|blockID=testSinkID|classURI=urn:
    java:de.uni_stuttgart.nexus.streamFederation.sinks.basic.fileOutputStream.
    FileOutputStreamSink|blockType=sink|remoteLocator=|&operation=|blockID=testOperationID|
    classURI=urn:java:de.uni_stuttgart.nexus.streamFederation.operators.basic.image.
    ShowImageOperator|blockType=operation|remoteLocator=|&queue=|queueID=testQueueID|queueURI=
    urn:java:de.uni_stuttgart.nexus.streamFederation.queues.basic.streamBlockingQueue.
    StreamBlockingQueue:2|&inputManager=|inputManagerID=testInputManagerID|blockID=
    testOperationID|inputManagerURI=urn:java:de.uni_stuttgart.nexus.streamFederation.
    inputManagers.basic.parallelAsync.ParallelAsyncInputManager|&commandId=|commandId=
    testCommand|&
```

Listing A.1: Encoded Setup Command

A.2 Setdown command

```
&commandType=|commandType=1|&confirmation=|confirmation=ok|&
```

Listing A.2: Encoded Setdown Command

A.3 Show Image Operator example

```java
package de.uni_stuttgart.nexus.streamFederation.operators.basic.image;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;
import java.io.InputStream;

import javax.microedition.lcdui.Image;

import de.uni_stuttgart.nexus.streamFederation.boxes.Box;
import de.uni_stuttgart.nexus.streamFederation.boxes.GraphicalResultOperation;
import de.uni_stuttgart.nexus.streamFederation.boxes.Operation;
import de.uni_stuttgart.nexus.streamFederation.boxes.impl.
    AbstracGraphicaltUnlinkedInputsOperation;
import de.uni_stuttgart.nexus.streamFederation.boxes.impl.AbstractOperation;
import de.uni_stuttgart.nexus.streamFederation.boxes.impl.AbstractUnlinkedInputsOperation;

public class ShowImageOperator extends AbstracGraphicaltUnlinkedInputsOperation {
        public ShowImageOperator() {
                super("Show image operator");
                // Set the inputs and outputs
                setInputIDs(new int[] { DEFAULT_INPUT });
                setOutputIDs(new int[] { DEFAULT_OUTPUT });
        }

        public boolean process(int inputID, Object data) {
                try {
                        System.out.println("Executing operation");
                        super.controller.showImage((byte[]) data);
                        System.out.println("START mando la imagen a guardar a fichero!");
                        send(data, DEFAULT_OUTPUT);
                        //data=null;
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return true;
        }

}
```

Listing A.3: ShowImageOperator class

A.4 Stream Blocking Queue example

```
package de.uni_stuttgart.nexus.streamFederation.queues.basic.streamBlockingQueue;

import java.util.Vector;

import de.uni_stuttgart.nexus.streamFederation.boxes.Queue;

/**
 * Implementation of a BlockingQueue
 *
 * @author Antonio Fernandez Zaragoza
 *
 */
public class StreamBlockingQueue implements Queue {

        private Vector vector;
        private int MAX_CAPACITY;
        private int currentCapacity;
        private boolean finalValue;
        private Object lock = new Object();

        public StreamBlockingQueue() {
                MAX_CAPACITY=Integer.MAX_VALUE;
                vector = new Vector();
                finalValue = false;
        }

        public StreamBlockingQueue(int maxCapacity) {
                if (maxCapacity>=1) {
                        MAX_CAPACITY=maxCapacity;
                } else {
                        MAX_CAPACITY=Integer.MAX_VALUE;
                }
                vector = new Vector();
                finalValue = true;
        }

        /**
       * Sets the capacity of the queue only if previous not default value
       * was not set
       *
       * @param capacity
       */
```

```
        public void setCapacity(int capacity) {
                if (!finalValue) {
                        MAX_CAPACITY = capacity;
                        finalValue = true;
                }
        }

        public synchronized void put(Object object) {
                if (vector.size()==MAX_CAPACITY) {
                        System.out.println("Put(): Blocking");
                        lock();
                }
                vector.addElement(object);
                synchronized (lock) {
                        lock.notify(); //
                }
        }

        public Object take() {
                if (vector.size()==0) {
                        System.out.println("Take(): Blocking");
                        try {
                                synchronized (lock) {
                                        lock.wait();
                                }
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                Object value = vector.elementAt(0);
                vector.removeElementAt(0);
                unLock();
                return value;
        }

        public int numElements() {
                return vector.size();
        }

        public synchronized void clear() {
                vector.removeAllElements();
        }

        /**
       * Retrieves and removes the head of this queue, waiting
       * if necessary up to the specified wait time if no elements
       * are present on this queue.
       *
       * @param timeout how long to wait before giving up
       * @return the head of this queue, or null if the specified waiting
       * time elapses before an element is present.
       */
        public Object poll(long timeout) {
                Object value = null;
                if (vector.size()==0) {
                        System.out.println("Poll(): Blocking for " + timeout + " ms");
                        try {
                                synchronized (lock) {
                                        lock.wait(timeout);
                                }
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                if (vector.size()>0) {
                        value = vector.elementAt(0);
                        vector.removeElementAt(0);
                        unLock();
                }
                return value;
        }
        /**
       * Inserts the specified element into this queue, waiting
       * if necessary up to the specified wait time for space to
       * become available.
       *
       * @param object the element to add
       * @param timeout how long to wait before giving up, in units of unit
       */
        public boolean offer(Object object, long timeout) {
                if (vector.size()==MAX_CAPACITY) {
                        System.out.println("Offer(): Blocking for " + timeout + " ms");
                        lock(timeout);
                }
                if (vector.size()<MAX_CAPACITY) {
                        vector.addElement(object);
                        unLock();
                        return true;
                }
                return false;
        }

        public int remainingCapacity() {
                return MAX_CAPACITY;
        }

        private void lock() {
                try {
                        synchronized (lock) {
                                lock.wait();
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

        private void unLock() {
                synchronized (lock) {
                        lock.notify();
                }
        }

        private void lock(long timeout) {
                try {
                        synchronized (lock) {
                                lock.wait(timeout);
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }

}
```

Listing A.4: StreamBlockingQueue class

A.5 File Output Stream Sink example

```
package de.uni_stuttgart.nexus.streamFederation.sinks.basic.fileOutputStream;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Enumeration;

import javax.microedition.io.Connector;
import javax.microedition.io.file.FileConnection;
import javax.microedition.io.file.FileSystemRegistry;

import net.jxta.util.java.net.URI;
```

```java
import net.jxta.util.java.net.URI.MalformedURIException;

import de.uni_stuttgart.nexus.streamFederation.boxes.Queue;
import de.uni_stuttgart.nexus.streamFederation.boxes.impl.AbstractSimpleSink;
import de.uni_stuttgart.nexus.streamFederation.queues.basic.streamBlockingQueue.
    StreamBlockingQueue;

public class FileOutputStreamSink extends AbstractSimpleSink {


        private URI filePath = null;
        private static final String DEFAULT_PATH = "foto";
        int fileNum = 0;

        public FileOutputStreamSink() {
                super();
        }

        public void setLocator(URI locator) throws IllegalArgumentException {
//              this . outputFile  = new File( locator );
                this.filePath = locator;
        }

        //@Override
        public void init() {
                if (getInputQueue() == null) {
                        setInputQueue(new StreamBlockingQueue(10));
                }
                setInitialized(true);
        }

//      @Override
        public boolean receive(Object o) {

                byte[] objectBytes;
                boolean received = false;
                objectBytes = (byte[]) o;
                try {
                        getInputQueue().put(objectBytes);
                        received = true;
                        System.out.println("SINK: Data received");
                } catch (Exception e) {
                        System.out.println("SINK: Data received no correct");
                        e.printStackTrace();
                        this.stop();
                }

                return received;
        }

        public void run() {
                // Whenever is an inputStream in the Queue, send it
                Queue queue = getInputQueue();
                byte[] buffer = null;

                while(shouldWork()) {
                        try {
                                //Wait and get data
                                buffer = (byte[])queue.take();
                                //Write file
                                writeInFile(filePath, buffer);

                                System.out.println("Writing finished: Closing file");
                                System.out.println("Writing finished: File Closed");

                        } catch (Exception e) {
                                System.out.println("START error general en fileoutputstreamsink");
                                e.printStackTrace();
                        }
                }

        }
        /**
        * Write data in a file
        *
        * @param configFilePath full path of file
        * @param data data to be stored
        * @return true if there was no problem in the operation
        */
        public boolean writeInFile(URI filePath, byte[] data) {
                FileConnection conn = null;
                //Get filePath
                if (filePath==null) {
                        try {
                                filePath = new URI(getDefaultPath(fileNum));
                                fileNum++;
                        } catch (MalformedURIException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println("START path of file: " + filePath);

            try {
                 conn = (FileConnection) Connector.open(filePath.toString(), Connector.
    READ_WRITE);
                 if (!conn.exists()) {
                   conn.create();
                 }
                 OutputStream outSt = conn.openOutputStream();
//             for (int j=24116; j<24146; j++) {
//                             System.out.println("data2: " +data[j]);
//                     }
                 outSt.write(data);
                 conn.close();
             } catch (Exception e) {
                 System.out.println("START error guardando con connection file");
                 closeConn(conn);
                 return false;
             }

             return true;
        }

        private String getDefaultPath(int fileNum) {
                Enumeration roots = findRoots();
                String root = "";
                String path = "";
                while (roots.hasMoreElements()) {
                        //Create path to file
                        root  = (String) roots.nextElement();
                        path = new String("file:///" + root + DEFAULT_PATH + fileNum
                                        + ".png");
                }
                return path;
        }

        private Enumeration findRoots() {
                return FileSystemRegistry.listRoots();
        }

        private void closeConn(FileConnection conn) {
                try {
                        conn.close();
                } catch (IOException e) {
                        //Ignored
                        e.printStackTrace();
                }
        }
}
```

Listing A.5: FileOutputStreamSink class