Master Project

Push To Chat

Realized by

The Quang Nguyen

Supervisor

Professor Jean-Yves Le Boudec (EPFL, LCA)

Assistant

Alaeddine El Fawal (EPFL, LCA)

Ecole Polytechnique Fédérale de Lausanne (EPFL)

Winter Semester 2007-2008

Contents

1 Introduction **7**

2 Development environment and Architecture **8**

2.1 Development environment 8

2.1.1 Hardware equipments 8

2.1.2 Language programming 9

2.1.3 Building MIDlet 11

2.2 Architecture 14

3 Application **16**

3.1 Functionalities 16

3.2 Java Micro Edition version 17

3.2.1 User Interface 20

3.2.2 Message reordering process 21

3.2.3 Displaying messages with CustomItem 26

3.2.4 Block or Unblock users 29

3.2.5 SLEF status 31

3.2.6 User's identification 31

3.3 Java Standard Edition version 32

4 Self Limiting Epidemic Forwarding (SLEF) Implementation **33**

4.1 Protocol description 33

4.2 Protocol implementation 34

4.2.1 The epidemic buffer 37

4.2.2 Reception 38

4.2.3 Transmission/Retransmission 42

4.2.4 Checklist 48

5 MAC Broadcast **49**

5.1 J2ME implementation 50

5.1.1 MACBroadcast 50

5.1.2 MACReceiver 50

5.2 J2SE implementation 51

5.2.1 MACBroadcast 51

5.2.2 MACReceiver 51

6 Deployment **52**

7 Conclusion and Future work **54**

Acknowledgements

I would like to say a special thanks to:

- **Professor Jean-Yves Le Boudec** (EPFL, LCA) for his supervision of this project.
- **Alaeddine El Fawal** (EPFL, LCA) for his kind guide during the whole semester.

Chapter 1

Introduction

Nowadays, communications using small, smart and attractive electronic devices become popular and take an important place in our everyday life. With the expansion of small mobile devices along with their improvements in performance, we would like to build an application that facilitates the communications between people. With the idea that each device is a mobile station, we would like to create a self-organized network, which also can be extendable with the host's movement. The future application would be useful for the traffic information, security message, or, simply it would offer the possibility to chat during train time travel.

The Self Limiting Epidemic Forwarding (SLEF) protocol designed by Alaeddine El Fawal provides all necessary mechanisms for building such an application. And this gives birth to the *Push To Chat* project.

The objective of this project is to build a chat application over SLEF for Smartphones.

Chapter 2

Development environment and Architecture

In this chapter, we would like to discuss about the development environment and the architecture of the application. By development environment, we mean the hardware equipments and the language programming. In the latter part, we will also give some general concepts and components of the language, without diving into details. Then, in order to understand the two next chapters, we aim to explain the interactions between different layers of the system.

2.1 Development environment

2.1.1 Hardware equipments

This project targets Smartphone field. It is worth noting that there are Personal Digital Assistant (PDA) phone and Smartphone on the market. In spite of their similar functionalities, their approaches are different. A Smartphone is a mobile phone with advanced functionalities of a classical computer, and a PDA phone is a complete PDA to which mobile phone's functionality is added. Hence, in general, when we compare the functionalities, the available memory and the processor power of these two devices, we can say that a Smartphone is a light version of a PDA phone.

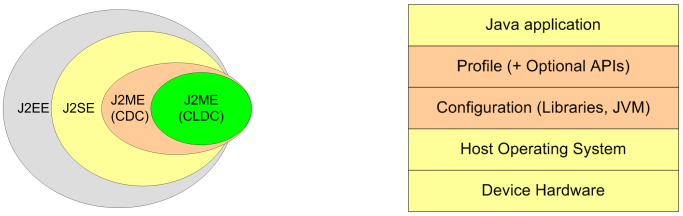
The chosen Smartphone for development is the HTC S620. The device runs on Microsoft ©Windows Mobile (version 5.0) platform with 64MB of RAM and 128MB of ROM. Its processor is the Texas Instruments OMAP 850 running at 201 MHz. Moreover, it also has advanced connectivity, such as Bluetooth ©2.0 and Wi-Fi ©IEEE 802.11. The IEEE 802.11 is the mandatory type of connection for this project. Although the HTC S620 is a non touch-screen device, it offers a compact QWERTY keyboard and a large QVGA display (320x240 with 65,536 colors).

2.1.2 Language programming

Java is the main programming language for this project. This language is known for its portability, available libraries, online tutorials and for its documentation. The official support center at Sun's website continuously maintains and updates the documentation. And, an incontrovertible advantage of Java comparing to other languages is its highly active community. At the Java's official forum [1], developers usually can find answer to a lot of technical problems.

However, although smartphones have many functionalities, they still are limited-resources devices. They are limited in screen resolution, in memory and in processor power. So clearly, we do not have enough performance nor memory to deploy the standard Java Virtual Machine (JVM) to run Java 2 Standard Edition (J2SE) application, and we need a lightweight version of Java.

The Java 2 Micro Edition (J2ME) is a Java platform which is designed for small devices. That can be a pager, mobile phone or a personal digital assistant (PDA). Actually, J2ME is not a proper subset of J2SE (hence nor of the Java 2 Enterprise Edition (J2EE)). Beside a lightweight virtual machine and a lightweight subset of the standard edition, J2ME contains additional packages specially designed for small devices, such as `javax.microedition.*` packages. Figure 2.1(a) illustrates the Java 2 platform architecture.



(a) Java Platform Architecture (b) J2ME Layers

Figure 2.1: The Java Platform Architecture and the J2ME Layers

J2ME is divided into *configurations*, *profiles* and *optional APIs*. A valid combination of a configuration and a profile targets a specific family of devices. Because the devices are limited in performance, these combinations are necessary in order to balance portability with performance and feasibility in the real world.

Configuration

A configuration specifies the Java Virtual Machine (JVM) and a set of supported Java libraries. Usually, based on memory contraints and the power of the processor, importing a specific configuration into devices is the responsibility of the device manufacturers. For instanct, there are two configurations:

[1] Java Technology Forums, http://forum.java.sun.com/index.jspa

Connected Device Configuration (CDC) This configuration is designed for devices with 2 MB or more of total memory and with advanced network connection. It targets devices like car navigation systems or high-end PDAs. CDC specifies the use of the full Java 2 virtual machine, which is called Compact Virtual Machine (CVM) in this context.

Connected, Limited Device Configuration (CLDC) This configuration is included in CDC. CLDC targets smaller devices (such as mobile phones) than those targeted by the CDC. It specifies a small JVM called Kilobyte Virtual Machine (KVM), due to the fact that the size is measured in kilobytes. As it has a small size, the KVM cannot support all JVM's features and has some security problems.

In its version 1.0, CLDC does not support floating point calculations at all. It was a big inconvenience, because that means there are no `float` or `double` primitive types, neither the corresponding wrapper types, `java.lang.Float` and `java.lang.Double`. Since CLDC 1.1, these types are included.

Profile

As illustrated in figure 2.1, a profile is implemented on top of a configuration and provides additional APIs, such as a graphical user interface, security, and network connectivity, in order to develop applications. There are several profiles in the market, such as PDA Profile, Personal Profile... but the two most important ones are Foundation Profile and Mobile Information Device Profile.

Foundation Profile (FP) This profile is based on the CDC. It provides a rich Java network environment. However, FP does not support graphical user interfaces (GUIs); Abstract Window Toolkit (AWT) and Swing packages are not present.

Mobile Information Device Profile (MIDP) This profile is based on the CLDC and is widely deployed on mobile devices. Even if other CLDC-based profiles have been mentioned, MIDP is the only well supported and available for instant.

As discussed earlier, each valid combination of a configuration and a profile targets a specific family of devices. The CDC/FP and CLDC/MIDP are the most typical combinations that we can find. A J2ME platform implementation can have only one configuration, but it can have multiple profiles that are built on top of one another. For example, Personal Basic Profile can be used in order to add GUI support on top of the FP. Besides, those valid combinations can be extended with optional packages which are built to support specific application needs (Mobile Media API supports audio/video controls and streaming media, Bluetooth API to supports Bluetooth communication protocol).

As we are interested in developing an application that can be deployed on different types of smart phones, CLDC/MIDP is adopted. MIDP applications are also called MIDlets.

CDC and CLDC implementations on the market

It is necessary to mention that there are required features and recommended features in J2ME specifications. For example, CLDC/MIDP 2.0 implementers must provide support for Http connection, but for other connection types, it's up to them to provide support or not. Sun Microsystems does not release any official implementation for CDC and CLDC. However, as the market of mobile devices grows up, some companies and independent developers are interested in providing solutions for J2ME. During this project, some solutions have been studied:

- **Intent Midlet Manager** (Tao Group) implements CLDC 1.0 and MIDP 2.0 and is a well-known solution. It comes pre-installed on HTC phones. However, the only connection type that it supports is HttpConnection type. DatagramConnection, SocketConnection and ServerSocketConnection are not supported. Moreover, the group does not exist anymore and there is not any official support for the product. One of the developers is still giving his help to the end-users on the private forum *xda-developers* [2]

- **CrEme** (NSIcom) is a CDC 1.0 implementation for Windows CE devices. NSIcom also provides additional packages for advanced graphics (Swing) and connectivity (Serial Communication). However, it is not compatible with Windows Mobile Smartphone Edition.

- **WebSphere Everyplace Micro Environment** (IBM) provides a very complete solution for both CDC and CLDC. DatagramConnection, required for this project, is also supported along with other connection types. The solution, which is commercial, comes with a good documentation.

  This is the solution that we use in this project. Further information about how to download trial version or how to buy the solution can be found at its official website [3].

2.1.3 Building MIDlet

We would like to discuss in this section some general components of the J2ME language. We intend to explain only components used during this project. Our goal is not to make the reader dive into details, but is only to give necessary information to understand how the application is built. We would like to recommend interested readers to refer to the complete J2ME book of Sing Li and Jonathan Knudsen [1] for more advanced knowledge about the J2ME language.

MIDlet Life Cycle

J2ME applications are real Java application that run under control of a Java virtual machine. But in reality, with devices like mobile phones, Java applications cannot be invoked

[2] xda-developers forum, http://forum.xda-developers.com/
[3] WebSphere Everyplace Micro Environment , http://www-306.ibm.com/software/wireless/weme

by command shell. Actually, the entire life cycle of a J2ME application is controlled by an Application Manager Software (AMS). This AMS also controls the installation, execution and the application removal.

The base class for all MIDP applications is `javax.microedition.midlet.MIDlet`. The main class of a MIDP application always extends this base class. MIDlets have a small set of well-defined states. Along with the constructor method, we can find in a MIDlet subclass three other methods `startApp()`, `pauseApp()`and `destroyApp()` which are called to perform transitions from one state to another state. Figure 2.2 shows the states of a MIDlet and the transitions between them.

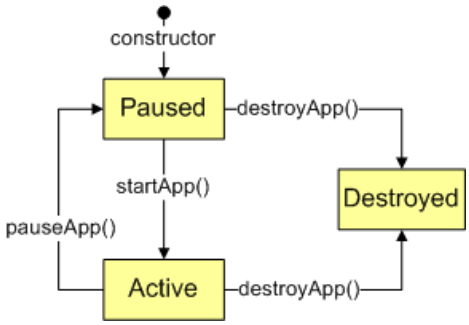


Figure 2.2: The J2ME MIDlet's life cycle

When the MIDlet is started, the AMS calls its constructor and the MIDlet enters to the *Paused* state. Then, only when the AMS calls the `startApp()` method that the MIDlet will be in *Active* state. From here, the AMS can suspend the MIDlet's execution and place it in the *Paused* state, and then put it back into *Active* state by calling `pauseApp()` and `startApp()`. After that, `destoyApp()` put MIDlet into the *Destroyed* state, where the application is stopped and awaits garbage collection.

If an active application wants to put itself to *Paused* or *Destroyed* state, it can call respectively `notifyPaused()` or `notifyDestroyed()` method.

User interface

One of the main differences between building a MIDlet and a classical J2SE application resides on the user interface. A MIDlet is usually built to run on different devices having screen of all sizes, with or without color. Moreover, all devices don't have the same input capabilities; one can have a numeric keypad while another possesses a touch screen. That's why the way that user interfaces are built is differently conceived compared to a `Swing` or `AWT` graphical interface. With MIDP, rather than assign a command to a specific place, we only can ensure that the command exists somewhere in the interface, and it is the role of the J2ME implementation to design and place it where it should be. That's the reason why depending on the J2ME implementation, when running the same MIDlet on different devices, the user interface may be different.

MIDP contains the user interface classes in the `javax.microedition.lcdui` package. The device's display is represented by an instance of the `Display` class whose main purpose

is to determine what is currently shown in the device screen. The content that can be displayed in the device screen is a `Displayable`. The difference between `Display` and `Displayable` is that `Display` class represents the display hardware, whereas `Displayable` is the content that can be shown on the display. In order to change the contents of the display, a `Displayable` instance is passed to the provided `setCurrent()` method of the `Display` class. As the name of the method can signify, only one instance of `Displayable` can be displayed at the same time. We cannot have a multi-windows interface as it is the case for a J2SE user interface.

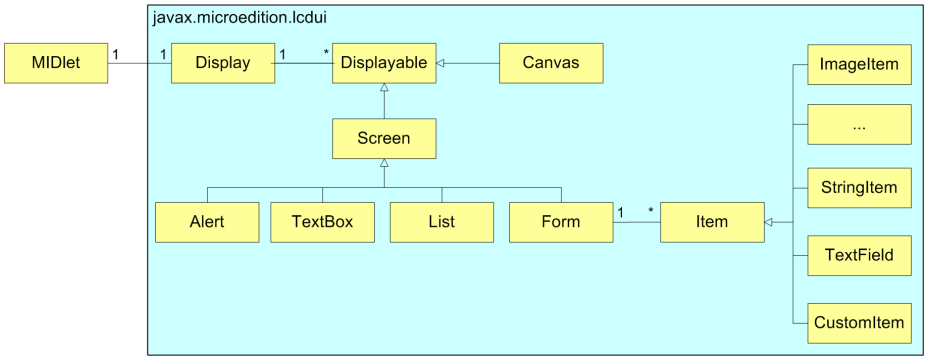


Figure 2.3: The J2ME's lcdui package

Figure 2.3 shows `Displayable`'s sub-classes. `Screen` and `Canvas` are two direct sub-classes which target different type of applications. `Screen` targets generic applications whereas `Canvas` targets game developments. The development of this project only uses `Screen`'s classes, as they already provide all necessary and convenient functionalities. `Screen` has four sub-classes: `Alert`, `List`, `Form` and `TextBox`. As mentioned earlier, only one instance of these four types of `Screen` can be displayed in the device screen at the same moment. The following will give a brief description of these types.

TextBox This is the simplest type of screen. It allows the user to insert or to modify a text.

Alert This is a screen containing text or image. It is used to show informative message, errors or exceptions. There are two types of alerts: *modal* alert and *timed* alert. A *modal* alert needs confirmation from user to be dismissed, while a *timed* alert doesn't need confirmation but will be shown for a certain amount of time.

Form This screen is used to create more advanced interface. It includes a collection of user-interface components, called *items*. They can be images, read-only text fields, editable text fields, editable date fields, gauges, choice groups, or customized items.

These items are all subclasses of `javax.microedition.lcdui.Item` class. There are eight items, but only `StringItem`, `TextField` and `CustomItem` are used in this project. The `StringItem` represents a text label. Whatever the text length can be, this item can still be conveniently displayed in the device screen. For that, the text will automatically be made scrollable if needed. However, the text's font cannot be easily customized. Then, the `TextField` represents an editable string. It can be used

for the user to compose his message. And the third item used in the application is `CustomItem`. As its name indicates, this item is customizable by sub-classing and is employed to introduce new visual and interactive elements to `Form`. For this end, its sub-classes have possibility to customize the size, colors, fonts and graphics of the item. They also handle events introduced by users by keys, pointer actions or traversal actions. The message area displaying incoming and outgoing messages of Push To Chat application is built using this item.

It is interesting to mention that `Form` provides a concept of *focus*, which refers to the currently selected item in the form. This gives possibility to perform some specific actions when a given item is selected and when the user invokes some command.

List This screen allows the user to select elements from a list of choices. The selection can be *exclusive*, *implicit* or *multiple*. As part of this Push To Chat application, this screen is used to propose a list of users to be blocked or unblocked.

Handling User Input with Commands

A `Displayable` (i.e. four screens discussed above) or an `Item` has possibility to contain a list of `Command`s in purpose of supporting a flexible interaction with users. A `Command` can be compared to a button in J2SE GUI and can be invoked for example by a keypad button, touch screen or voice recognition.

In order to respond to an event invoked by an user, the `Command`'s container (`Displayable` or `Item`) has to register an object called *listener*. Otherwise, even if commands are shown in the device screen, nothing will automatically happen when an user invoke one of them. The listener is an object that implements the `CommandListener` interface by defining a single method `public void commandAction(Command c, Displayable s)`. Within this method, specific actions are defined when a `Command` represented by c is invoked. Then, to register the listener to its container, the container's method `setListener()` is called.

2.2 Architecture

The main application will allow end-users to exchange text messages over an ad-hoc network. We adopt the layer approach to present the architecture of the system. There are three layers: (1) Application, (2) SLEF and (3) MAC. The layer approach has several important advantages. It allows having a clear structure and as it is the case in the traditional network, each layer can be developed and maintained independently. The development or maintenance of one layer does not affect the operation of other layers. Moreover, SLEF was designed for different types of application, and not only for chat application. We will present here the role of each layer and how they communicate to each other. The two first ones will be discussed in detail in the two next chapters.

Application The Push To Chat application itself is built at this layer. It is responsible for the interaction with end-users. Its two essential functions are (1) executing user's commands and (2) displaying messages. For this mean, a graphical user interface

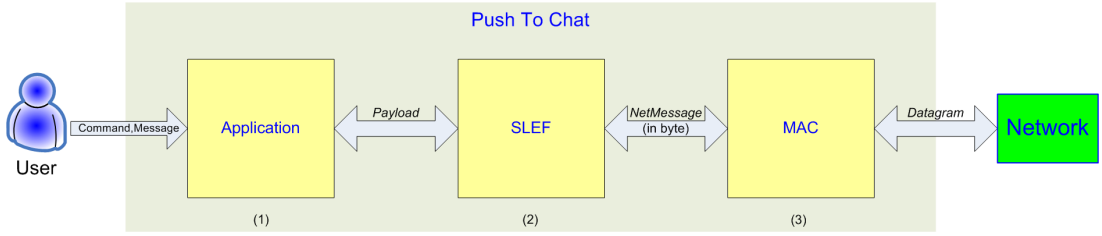


Figure 2.4: Architecture

(GUI) was provided. Packets exchanged between the application layers are referred as `Payload`s. This unit carries the message to display and the message's information, such as the originator and sequence number of the message.

SLEF This is the layer at which the protocol SLEF is implemented. It resides under the application layer and above the MAC layer. We also adopted the concept of encapsulation in this layer. Additional information, such as the MAC address and the age of the packet, will be added to payload that is passed by the application layer. The result of this encapsulation will form a `NetMessage` which is the unit exchanged between SLEF layers.

MAC The role of this layer is to send and receive packets over the network using the pseudo-broadcast mechanism of the SLEF protocol. This mechanism is included in the *Efficient use of MAC broadcast* function and is not studied in this project. Instead of it, we use an UDP connection. It transmits packets passed by the upper SLEF layer using the broadcast address and when it receives a datagram from the network, it will deliver to that SLEF layer again. The unit used for communication between these two layers is an array of bytes containing the content of a NetMessage. Datagram is directly constructed from this array without any modification. As a perspective, we still call this layer *MAC* as the expected pseudo-broadcast mechanism will be built at this layer.

Chapter 3

Application

In this chapter, we will present in detail the Push To Chat application. First, we will discuss about the functionalities that the application is supposed to provide, and then the manner that they are implemented. We will provide at the end two versions of the application. The first one is implemented in J2ME for being deployed to Smartphones and the second one is in J2SE for traditional computers.

3.1 Functionalities

1. As the application is designed for chatting, the application must provide a graphical user interface (GUI) containing (1) a messages area for displaying incoming and outgoing messages and (2) an input text field allowing the user to compose the outgoing text.

2. The newest message (last received message) is supposed to be distinctly displayed. For example, this message can be displayed in bold and/or with a different color.

3. The messages area must be scrollable. As the number of messages increases, the messages area cannot display all of them. It would be regrettable if end-users cannot read other hidden messages.

4. At this application layer, we do not have any guarantee for the messages of the same source to be delivered in the right order. As the application is built on top of SLEF layer, the transmission and retransmission of messages depend on the SLEF state at each node in the network. Moreover, we use UDP connection for broadcasting and receiving packets, thus either the packets reception or their deliverance sequence is guaranteed. Messages may arrive in a wrong sequence, and the application must provide a mechanism to reorder them. The mechanism handling this problem is called *message reordering* and will be explained in section 3.2.2.

5. The application also aims to demonstrate the mechanism of SLEF protocol. Showing the current state of the lower SLEF layer while the application is running is one solution.

6. Messages are broadcasted to all nodes in the network, so everyone can compose messages and can send to all others. Unfortunately, undesirable messages exist. Since we are in a distributed network, and in a decentralized system, there is not possibility to forbid an user to send messages. But a local user can stop receiving messages of one or more specific users, so that these messages will not be displayed on his screen. This process is called blocking/unblocking users.

3.2 Java Micro Edition version

The first version of the application was developed in J2ME for Smartphone. To present the application, we will start from a global view (for example how the program runs) to very specific problems (for example how the message reordering problem is solved). Figure 3.1 shows the lifecycle of the application.

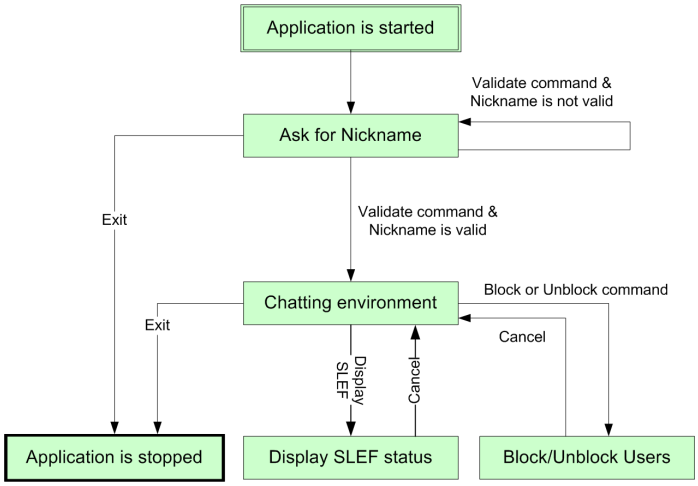


Figure 3.1: The J2ME application's life cycle

Once the application is started, it will ask for the nickname that the user wants to use for chatting. While this value is empty, the application will keep asking. Otherwise, the application will enter to the chat environment, where the local user can compose his messages and receive messages from other users participating to the same network. At this state, user has three other possibilities. The first one displays the current status of SLEF layer, the second one allows the local user to block or unblock other users, and the last one quits the application. In the first two cases, if the local user wants to stops the application, it has to come back to the chat environment before executing the *Exit* command.

The provided package `PushToChat.Application` contains all necessary classes for this application part. We give below a description for each of these classes:

PushToChatMidlet.java This is the main class of the Push To Chat application. It is responsible for almost functionalities listed in section 3.1 above, that is to say the graphical user interface, the message reordering process, the SLEF status displaying

and the blocking/unblocking users process. It is straightway to mention that the user's commands are also handled in this part.

Payload.java This represents the unit used to exchange information between application layers. As illustrated in figure 3.2, a payload contains the user's MAC address, its nickname and also its message. `receivedFrom` indicates the node from which this payload is received. If it differs from the user's MAC address, then the payload has been forwarded by another node, which is different from the payload's originator. This is one of the advantages of the SLEF protocol and the information will be displayed on the device screen. The class also provides a method, called `getBytes()`, for converting the current state of the object into an array of bytes. This process is required when the payload needs to be sent over the network, and then needs to be reconstituted. It can be compared to a serialization. Actually, this is done "manually" because J2ME does not provide the interface `Serializable`. Then, as this class contains the message, it also provides a method returning the String to be displayed in the screen's device.

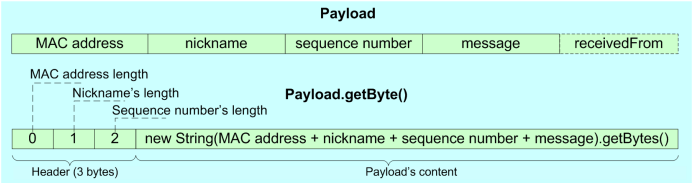


Figure 3.2: Payload

User.java This class represents an user in the network, at the application layer view. It contains information related to an user such as its nickname, its MAC address and also the payload with the lowest sequence number that has been received by the local user, and it also contains a wait list indicating the next expected and the delayed payloads. This aspect belongs to the message reordering mechanism and will be discussed in section 3.2.2.

DisplayItem.java This class is responsible for functionalities 2 and 3 listed above. It issues a scrollable text box used to display the incoming and outgoing messages with the possibility to repeat the scrolling movement. Then, the newest message is displayed in bold and in blue, in order to distinct with other previously received messages. As J2ME does not provide convenient tools for doing all this, we will discuss about this `DisplayItem` in more detail in section 3.2.3.

KeepScrollingTask.java This class is a `TimerTask`, which is a task that can be scheduled for one-time or repeated execution by a `Timer`. In order to repeat the scrolling movement when the specified key is help down, an object of this class is periodically called by a timer initiated in `DisplayItem`.

The application can be divided into two main processes: ***Reception*** and ***Sending***.

The reception process is illustrated in figure 3.3, as a sequence of events. In the figure, italic words indicate methods in the corresponding class, while the label above the arrow indicates the method's parameter.

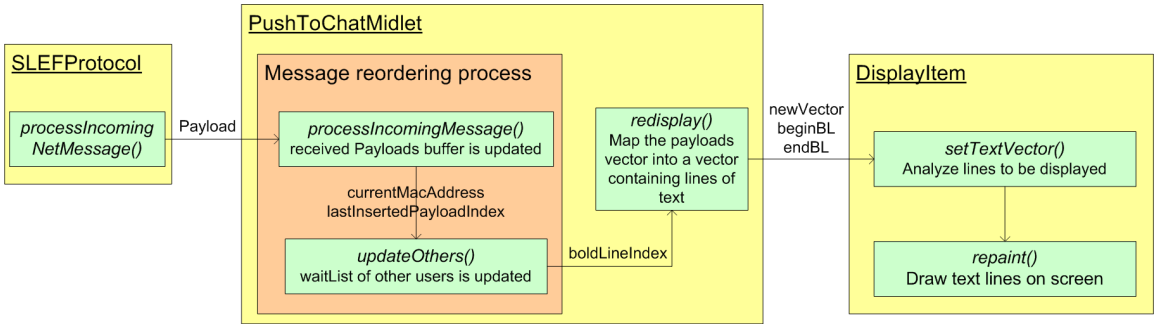


Figure 3.3: J2ME Reception

The reception process starts when the SLEF layer delivers a new received `Payload` to the application (i.e. to `PushToChatMidlet` class). It will enter into the message reordering process, which will be explained in detail in section 3.2.2 below. We use a buffer to store all received payloads in the reversed order that they are displayed in the device screen, called `receivedPayloadsBuffer`, which is a `Vector`. At the end of the message reordering process, the received payload will be inserted to the correct position to the buffer. As we intend to display this payload with a different font, the argument `boldLineIndex` is used to store the index of this payload in the buffer. Then we will map `receivedPayloadsBuffer` into a new buffer which contains this time the exchanged messages extracted from payloads. The new buffer is called `receivedMessagesVector`, which also is a `Vector`. Nevertheless, each entry of `receivedMessagesVector` corresponds to a line of text displayed on the device screen, and it is possible that an exchanged message needs to be mapped to several lines. Then, this buffer along with the indexes of the first and the last line to be displayed with a different font will be passed to the `setTextVector()` method of `DisplayItem`, which is responsible for displaying incoming and outgoing messages. As the number of messages increases, the device screen may become too small to be able to show all messages. `setTextVector()` method analyzes the portion of text lines that will be displayed on the device screen and then calls `repaint()` method in order to "paint" on the screen the corresponding text lines.

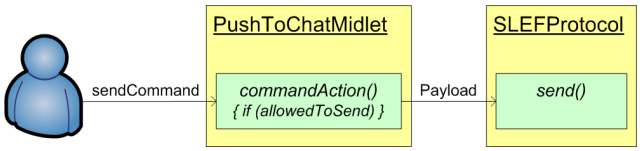


Figure 3.4: J2ME Sending

At the other side, the sending process is much simpler. Figure 3.4 illustrates this process. When the user invokes the specific sending command, called `sendCommand`, the listener will call `commandAction()` method to execute the specific actions. As SLEF protocol provides a *Congestion Control* mechanism which controls the sending flow of the application, the variable `allowedToSend` indicates whether the application can inject a new payload to

SLEF layer. Then, if it's authorized, a new payload containing the composed message will be created and injected to the lower SLEF layer. The payload is identified by its originator and its sequence number. The application has a global variable called `seqNum` initiated by 1 when the application starts. This variable is increased by 1 whenever a payload is successfully sent. Hence, the first payload has sequence number equal to 1.

3.2.1 User Interface

In order to build an user interface, notions discussed in section 2.1.3 are used. The main class `PushToChatMidlet` extends the base class `javax.microedition.midlet.MIDlet`. Along with its constructor method, `PushToChatMidlet` inherits three other methods (`startApp()`, `pauseApp()` and `destroyApp()`) which are called by the AMS to transit from one state to another state within the MIDlet's life cycle (section 2.1.3). Moreover, to respond the events invoked by the user, this class implements the `CommandListener` interface by defining the method `commandAction()`. In this method, we specify the sequence of actions to be executed when the local user invokes a specific command.

As discussed, when the application is started, the AMS calls `startApp()` method for the application to enter into the *Active* state. In this method, we initialize global variables such as the sequence number `seqNum` or the buffer containing received payloads `receivedPayloadsBuffer`, or also variables used for the GUI like the main container `Form` called `form`, or the `TextField` named `nicknameField` which allows the user to insert his nickname to. Furthermore, in order to access to the device's display, an instance of the `Display` class, called `display`, is also created by calling the `Display` static method `getDisplay()`. We create next an instance of the SLEF protocol class. Then, calling `askNickname()` method will guide the application to its first state for asking the user's nickname.

Referring to figure 3.1, which illustrates the application's life cycle, the main `Form form` will be used at two different states, *Ask for nickname* and *Chat environment*. Whereas, another `Form` called `statusForm` will be used for displaying the status of the lower SLEF layer and at last, a `List` called `displayableUsersList` will be employed to display a list of users to be blocked or unblocked. We would like to remind that only one `Displayable` object can be shown on device screen at the same time. We can set the next one to be shown by giving it as the argument of the `display.setCurrent()` method. A collection of `Commands` have also been defined, such as `sendCommand` for sending, `exitCommand` for quitting the application or `validateCommand` for validating an action. These `Commands` can be associated to a `Form` by its `addCommand()` method.

Figure 3.5 shows two screenshots of the application running on the HTC S620 Smartphone. The corresponding J2ME components are also illustrated in the figure. The first screenshot (a) shows the application in its *Ask for nickname* state and the second (b) in its *Chat environment* state. The `askNickname()` method prepares the first state. It adds to `form` two commands `validateCommand` and `exitCommand`, then adds a `TextField` called `nicknameField` for the user to compose his nickname. If a valid nickname is given (i.e. the nickname is not empty), the second state can be reached after calling the `initFormsForChat()` method and then the `showChattingArea()` method. Primar-

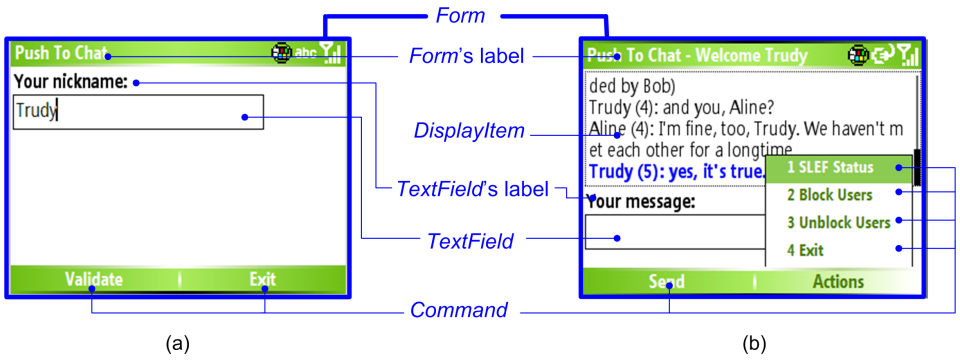


Figure 3.5: Push To Chat, J2ME's user interface

ily, `initFormsForChat()` method sets new values for the main form, and then it initializes `statusForm` and `displayableUsersList`. The main form, which is currently used for showing the nickname field, will be reinitialized for the chat environment. The main form will now contain a `DisplayItem` for displaying messages and a `TextField` called `outgoingMessage` for the user to compose a message. It also keeps another list of commands, such as `sendCommand`, `blockUsersCommand`. Once the main form is prepared, the sole instruction `display.setCurrent(form)` in the `showChattingArea()` method will allow the new form to be displayed next on the device screen.

Initializing `statusForm` for displaying the SLEF layer status is similar and will be discussed in section 3.2.5. However, there's a slight difference for initializing the `List displayableUsersList` for blocking or unblocking users, that we will discuss in section 3.2.4.

3.2.2 Message reordering process

As payloads may arrive out of sequence, and it is usually the case, this process provides a mechanism to sort incoming payloads and put them into the correct position in the buffer, whose entries are displayed on the device screen. This process is first proposed by Ibrahim El Ghandour[4] and is optimized in this project.

The whole process is included in the `processIncomingMessage()` and the `updateOthers()` methods of the `PushToChatMidlet` class. They also summary the two main parts of the process. The first part starts as soon as the SLEF layer delivers a payload to the application. In this part, the process analyzes the position for this payload to be inserted into the global buffer. This position is stored in a variable called `insertionPosition`. Then, in the second part, the process will update, if necessary, the position to be inserted of other awaiting payloads belonging to other users.

The process employs a global buffer, called `receivedPayloadsBuffer`, which is a `Vector`, to store all received payloads in the reversed order that they are displayed in the device screen. The first payload stored in the buffer, thus at position 0, will be displayed at the bottom of the screen and the last payload at the top of the screen.

The `User` class represents an user in the network. Besides information related to its identity,

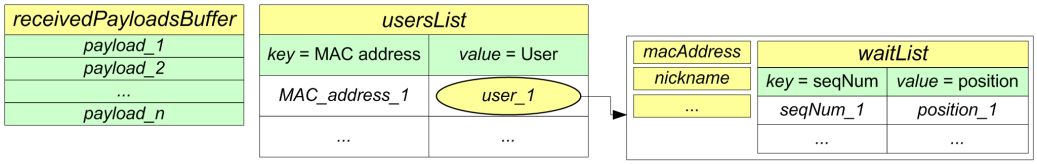


Figure 3.6: Three buffers of the message reordering process

each user maintains a wait list called `waitList`, which is a `Hashtable`. The table's keys correspond to the sequence number of next payloads to come, and its values indicate the position to be stored in the `receivedPayloadsBuffer` of each payload. For example, if `waitList` contains the pair (5, 10), it means that when the payload with sequence number equal to 5 is received, it will be inserted into the position 10 in the buffer. There's a particularity that for payload whose the entry in `waitlist` has a value equal to -1, it will be inserted at the first position in the received payloads buffer.

Along with this `receivedPayloadsBuffer`, the process also maintains another buffer containing participating users, called `usersList`, which is a `Hashtable`. The presence of an user in this table signifies that the application has received at least one payload originated by this user. The table's keys are the MAC address of the user's device, and its values refer to the `User` objects.

An important component of this process is the integer `incrementCredit`, which is a global variable. It stores the accumulated credits that the position of payloads in the users wait lists must add while being recovered or updated. Using this variable allows to postpone, when possible, the update of the payload's positions in the wait list of all users in the buffer. The update occurs only when the received payload is not stored at the first position in the received payloads buffer. Otherwise, the update is avoided and `incrementCredit` is increased by 1. This value will be taken into account at the next position's recovery or update. By this way, we can reduce significantly the number of loops while recovering all users in the buffer and their wait lists. The use of a loop is always very costly in term of performance.

The process is illustrated by figure 3.7. Some additional terms are used in the figure and need to be described. `seqNum` refers to the sequence number of the received payload, while `currentLowestSeqNum` refers to the smallest sequence number that the application has received from this user. And, `nextExpectedSeqNum` is the sequence number of the next expected payload, which is supposed to come next. In the wait list of the user, this payload is assigned to -1 and will be stored at the first position in `receivedPayloadsBuffer`.

The first part of the process starts at the payload's reception. It analyzes the payload and following cases are possible:

> ***Blocked user***: This user is blocked by the local user; hence its payloads will not be analyzed nor be displayed on the screen. The application maintains a list of blocked users, called `blockedUsersList`, which is a `Hashtable`. The MAC address of the user is stored as the table's key and its presence in this table implies that the user is blocked. The `return` instruction is called to quit the process.

Unblocked user: The MAC address of the user is not in `blockedUsersList` and the process continues.

> ***New user***: This is the first payload received for this user. Its MAC address is not present in `usersList`. This payload will be stored at the first position in `receivedPayloadsBuffer`, hence `insertionPosition` is equal to 0. An instance of `User` is created and is added to the list of users `usersList`. Within the `User`'s constructor method, an entry will be added to the wait list of this new user. This entry indicates to the next expected sequence number, which is equal to the current sequence number plus 1, and the corresponding position to be inserted, which is -1.
>
> ***Registered user***: `usersList` contains the user's MAC address, it means that the application has received at least one payload from this user. The `User` object and the wait list `waitList` are recovered. The process verifies whether `waitList` contains the payload sequence number.
>
> > ***Expected payload***: This payload is in the wait list and its assigned position is -1. This payload will be stored at the first position in the received payloads buffer (`insertionPosition` is equal to 0). The current sequence number will be removed from the wait list, and the next expected one will be added. The new entry in the wait list is (seqNum+1, -1).
> >
> > ***Delayed payload***: This payload is in the wait list and its assigned position is `p` $\neq$ -1. This payload will be stored at the position `insertionPosition = incrementCredit + p` in the buffer. Like in the previous case, the current sequence number is removed from `waitList`. However, not any additional entry is added to `waitList`, and the position of payloads whose sequence number is smaller than this current one will be increased by (`incrementCredit + 1`). This action corresponds to an implicit *shift down* movement in the received payloads buffer, and is the main mechanism of the process.
> >
> > The three following cases happen when this payload is not in the wait list, thus there is no pre-computed position for it. In order to compute the position for this payload, the process will base on previous received payloads. As mentioned earlier, each user keeps track of the received payload with the smallest sequence number. The first action is to verify whether the current sequence number `seqNum` is smaller the lowest stored one.
> >
> > ***New lowest payload***: This sequence number of this payload is smaller the lowest recorded one, labeled `currentLowestSeqNum`. We will first search for the position `n` of the latter payload that is currently stored in the received buffer. Then, the `insertionPosition` will be equal to (`n + 1`), in order to display the received payload after the current lowest one. Then, entries with keys equal to sequence numbers between `seqNum` and `currentLowestSeqNum`, and with values equal to `n` will be added to the user's wait list. And finally, the received payload is recorded as the lowest received one of the user.
> >
> > ***Low payload***: This payload has the sequence number greater than the lowest stored one, but smaller than the `nextExpectedSeqNum`. In this situation, we will looking for the position `n` in the received payloads buffer of the first

payload whose the sequence number is lower than the current one. Then, `insertionPosition` will be equal to `n`.
This case happens because we do not keep all awaiting sequence numbers in the wait list, but only some of them. This will be discussed right after, in *Newest payload* case.

Newest payload: The sequence number of this payload is greater than all stored in the user's wait list. As a result, it will be stored at the first position in the buffer (`insertionPosition = 0`). Then, we would like to keep at most 4 entries in the wait list, one for the next expected payload (same as in the *Expected payload* case), and three others are for payloads whose sequence number is at most 3 smaller than `seqNum`. Among these three entries, if one of them already exists, value will be increased by (`incrementCredit + 1`); otherwise, a new entry whose value equal to 1 will be added.

The two last cases are quite particular. If, in *Newest payload* case, instead of keeping only 4 entries in the wait list, we kept all of them, then the previous *Low payload* case would not have happened. The reason is that we would like to keep a good balance between performance and memory space. In the *Low payload* case, a search in received payloads buffer requires a loop, hence the performance will be panelized. However, keeping all entries in the wait list would cost the memory space, especially if these payloads never arrive. As a result, this method would make a good trade-off for performance and memory space.

At this state, the received payload is correctly inserted into the received payloads buffer at the position `insertionPosition` and the wait list of the user is also updated. So, the first part of the process ends with the call of `updateOthers()` method, which starts the second part. The purpose of this part is quite simple; it is to update the wait list of other users, if necessary. First, the update can be avoided if the payload is inserted at the first position of the buffer (`insertionPosition` = 0). In this case, `incrementCredit` will be increased by 1. Otherwise, the update occurs and each `User` object referred by the `usersList` will be recovered. Then, its `waitList` will be updated. The process increases by 1 the position of payloads in this wait list, if this position is greater than the insertion's position `insertionPosition`.

With this, the process ends and results (1) the correct insertion of the received payload in to the buffer, (2) the wait list of all users or their credit is updated and (3) the index of the payload to be displayed in bold, which is equal to `insertionPosition`.

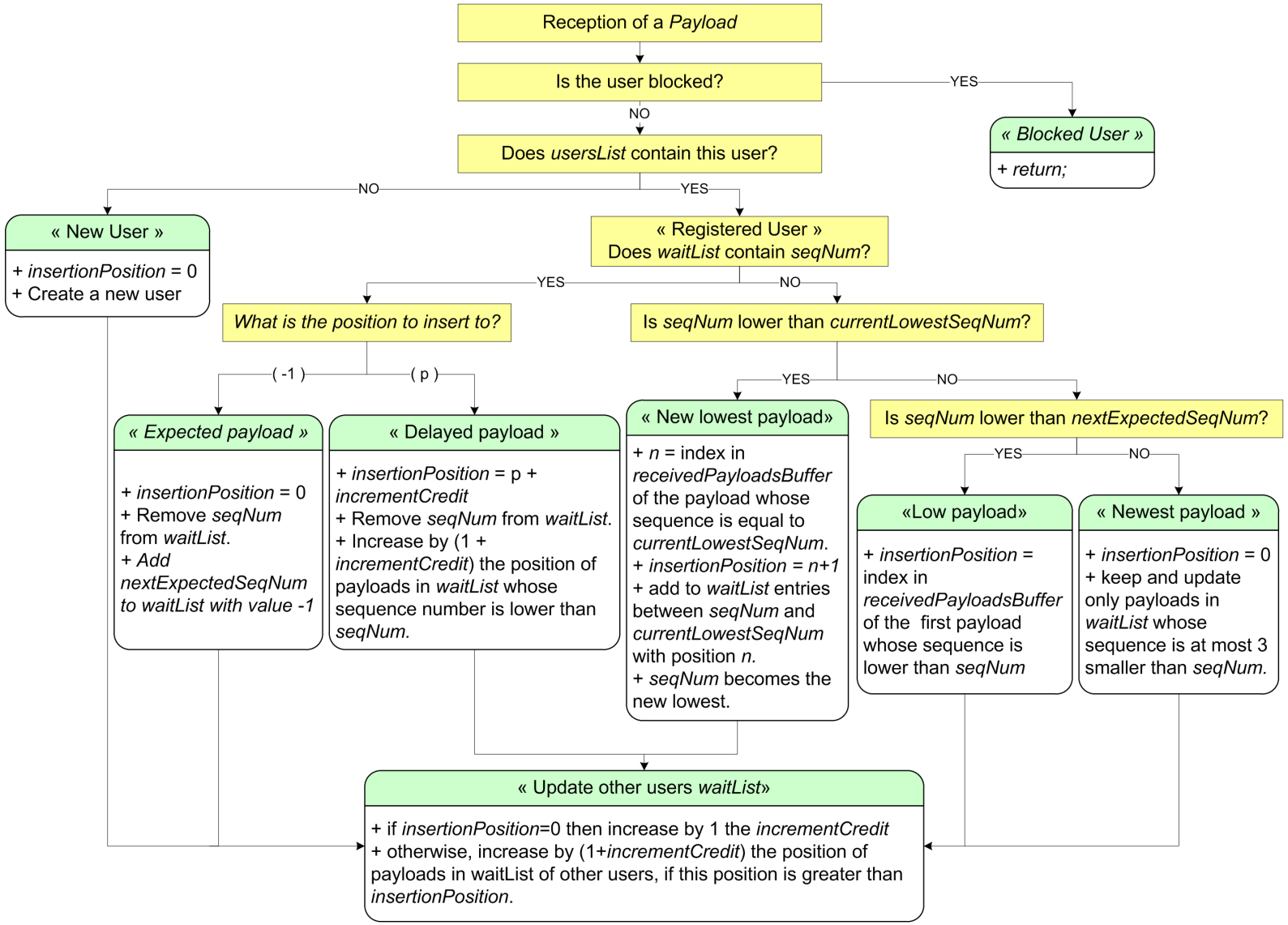


Figure 3.7: The messages reordering process upon reception

3.2.3 Displaying messages with CustomItem

At the end of the message reordering process, payloads in `receivedPayloadsBuffer` needs to be redisplayed and the newest one, whose position is equal to `insertionPosition`, is supposed to be differently displayed. In figure 3.3, we can see that the redisplaying process implicates three methods: `redisplay()` of the `PushToChatMidlet` class, then `setTextVector()` and `repaint()` belonging to the `DisplayItem` class. Figure 3.8 illustrates some important components of this `DisplayItem` class.

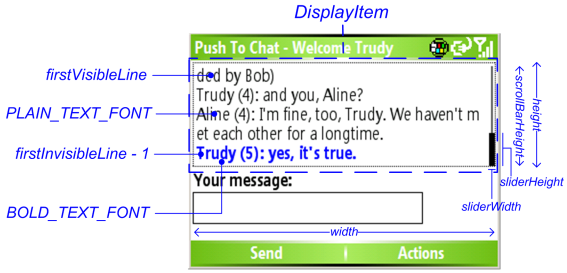


Figure 3.8: Push To Chat, J2ME's DisplayItem

The first method, `redisplay()`, is called right after the message reordering process. Its task is to map `receivedPayloadsBuffer` into a new buffer containing this time text lines of exchanged messages extracted from received payloads. The new buffer also is a `Vector` and is called `receivedMessagesVector`. Each element of receivedMessagesVector now corresponds to a line of text displayed on the device screen, and it is possible that an exchanged message requires several lines to be entirely displayed (e.g. message *"Aline (4): I'm fine,..."*) in figure 3.8).

If we consider the message as a `String`, then the necessary number of lines to display this `String` is equal to the number of characters in the `String` devided by the number of characters that the device can display on a line, the all plus 1. In our case, the unit used for the computation is not the number of characters, but is the width of each character displayed on the screen. Moreover, this width varies depending on the font that is currently used. The `Font` class of the package `javax.microedition.lcdui` provides necessary tools for this. We define two fonts in `DisplayItem` class, one called `PLAIN_TEXT_FONT` for painting the text in plain style and the other called `PLAIN_TEXT_FONT` for painting it in bold style.

```
Font BOLD_TEXT_FONT = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD, Font.SIZE_MEDIUM);
Font PLAIN_TEXT_FONT = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN, Font.SIZE_MEDIUM);
```

After extracting the message from each payload in the received payloads buffer, `redisplay()` method will call `getLineIndices()` method in order to compute the necessary number of lines for this message. This method has three parameters: `text` is the original message, `currentFont` corresponds to the font being used, and `lineWidth` is the available width for displaying a line of text. The value for the last parameter is somehow equal to `width - sliderWidth - someOffset`. This method returns a `Vector` whose the size plus one indicates the required number of lines, and each element in the vector stores the index of

the last character of the corresponding line in the original message. For example, if the vector contains {4,8}, it means that we need 3 lines for this message, the first line starts at 1st character to the 4th, then the second from the 4th to the 8th, and the last one from the 8th to last character of the string. Then, within the `redisplay()` method, each new line will be added to the `receivedMessagesVector`. When processing the payload which is supposed to display in bold (i.e. its position is equal to `boldLineIndex`, which also is equal to `insertionPosition`), the method will record the first line index to `beginBoldLine` variable and set the font to bold. Then, when all necessary lines are added, the method will record the last line to be displayed in bold, to `endBoldLine` variable.

```
for (int i = receivedPayloadsBuffer.size() - 1; i >= 0; i--) {
  // first line diplayed in bold
  if (i == boldLineIndex) {
    currentFont = incomingDisplayItem.BOLD_TEXT_FONT;
    beginBoldLine = receivedMessagesVector.size();
  } ...
  if (i == boldLineIndex) { // last line diplayed in bold
    endBoldLine = receivedMessagesVector.size();
  }
}
```

This will end the `redisplay()`'s task. Now that the `receivedMessagesVector` and indexes are prepared, `DisplayItem`'s `setTextVector()` method will be next called to compute the portin of text lines to be displayed. There are some components in this `DisplayItem`'s class that we would like to describe first.

`messagesVector`: this `Vector` is the same as the `receivedMessagesVector` passed by `redisplay()` method.

`maxVisibleLines`: The maximal number of lines that can be displayed in the text area. It is equal to the height of the area `height` minus some offsets, then the all divided by the height of a character displayed with a given font.

```
fontHeight = BOLD_TEXT_FONT.getHeight();
maxVisibleLines = (int) (height - yOFFSET)/fontHeight;
```

`firstVisibleLine`: The index of the first element in the messages vector `messagesVector` to be displayed.

`firstInvisibleLine`: The ending index of the element in the messages vector to be displayed. We stop displaying text line just before this index.

With the given arguments `beginBoldLine` and `endBoldLine`, we compute the number of lines to be displayed in bold and store it to `numOfBoldLines`. From here, we distingue three cases:

1). If the number of bold lines exceeds the maximal visible lines, then the first line to display is the first line that is supposed to be in bold. And we display only a number of lines equal to the maximal visible lines authorized for this area, which is `maxVisibleLines`. As a result, there will be hidden text line(s) at the bottom of the screen.

```
if (numOfBoldLines > maxVisibleLines) {
  firstVisibleLine = beginBoldLine;
  firstInvisibleLine = firstVisibleLine + maxVisibleLines;
}
```

2. The second case is when this number is smaller than `maxVisibleLines`, and the last line which is supposed to be in bold corresponds to the last line stored in the buffer. This case happen when the received payload is stored at the first position in `receivedPayloadsBuffer`, this payload will be displayed at the bottom of the screen.

```
else if (endBoldLine == messagesVector.size()) {
  firstInvisibleLine = endBoldLine;
  firstVisibleLine = firstInvisibleLine -  Math.min(maxVisibleLines, messagesVector.size());
}
```

3). The last case is when the portion of lines in bold is somewhere in the middle of the vector, and it does not exceed `maxVisibleLines`, then we try to central this portion on the device screen.

```
else {
  firstVisibleLine = Math.max(0, beginBoldLine - (maxVisibleLines - numOfBoldLines)/2);
  firstInvisibleLine = Math.min(firstVisibleLine + maxVisibleLines, messagesVector.size());
  // re-adjust the firstVisibleLine
  if ((firstInvisibleLine-firstVisibleLine) < maxVisibleLines) {
    firstVisibleLine = Math.max(0,
                firstVisibleLine - (maxVisibleLines-(firstInvisibleLine-firstVisibleLine)));
  }
}
```

Then, we call `repaint()` method to refresh the screen. This method will signal to the system that the item needs to be drawn again. In response, it will call the `paint()` method. We pass to this method a `Graphics` object, called `g`. This object represents the drawing surface and provides methods for drawing images and texts. We use its `setFont()` and `setColor()` methods for customizing the text, and use its `drawstring()` method to draw the text line stored in the buffer. As we have computed all necessary indexes, we extract all messages from the buffer `messagesVector` and draw them in the defined order and with the corresponding font.

The last thing to be drawn is a scrollbar. The `drawScrollBar()` method is responsible for this. It also uses methods of the `Graphics` class for drawing. The scrollbar slider's dimensions are re-computed each time the screen is redrawn. The slider reflects the size and the position of the portion of text lines that are being shown regarding to the size of the whole buffer `messagesVector`.

A scrollable area

Displaying messages is not the only task handled by this `DisplayItem` class. This class also provides a mechanism for making its display area scrollable.

At this state, a scrollbar is drawn. However, as the command events are not handled yet, no action will be taken to respond to user's request. The `CustomItem` class supports some input modes, including key pressing, key releasing and key repetition. The last mode occurs when a key is pressed down for a long period. When a key is pressed, the system calls `CustomItem.keyPressed()` method, and when it's released, the `CustomItem.keyReleased()` method is called. As `DisplayItem` extends this `CustomItem` class, we will overwrite these two methods in order to define specific actions for our display area.

For reminding, a `Form` can be considered as a container of `Item`s and it also supports the concept of *focus*. A focused item is the item that is being selected by the user. When the user traverses into the item for the first time, a call to `CustomItem.traverse()` will occures. Then when the user traverses out this item, `CustomItem.traverseOut()` will be called. We redefine these two methods and use a `boolean`, called `isFocused` to indicate whether the `DisplayItem` is currently selected. Its value becomes *true* within `traverse()` and *false* within `traverseOut()`.

Afterwards, we also define the two other `boolean`s, named `keyUpPressed` and `keyDownPressed` to indicate if the "*Up*" key or the "*Down*" key is currently pressed. The numpad key 1 is used as the "*Up*" key and the numpad key 4 as the "*Down*" key. They correspond to constants `Canvas.KEY_NUM1` and `Canvas.KEY_NUM4`.

When "*Up*" key is pressed and the item is focused, `keyUpPressed` will becomes *true* and the `scrollTextDown()` is called. An "*Up*" key implies the text to be scrolled down, in order for the upper hidden text lines to be displayed. Hence, in `scrollTextDown()`, we decrease the values of `firstVisibleLine` and of `firstInvisibleLine`, if the first one is greater than 0. Then, we call `repaint()` for the item to be drawn again.

The same principle is applied when the "*Down*" key is pressed, `keyDownPressed` will becomes *true* and the `scrollTextUp()` method is called. This time, we increase those two indexes, if `firstInvisibleLine` remains smaller than the last text line's index, i.e. `messageVector.size()`.

Up to this state, the user can scroll up and down the area. But it has to keep pressing and releasing the key to scroll the area line by line. We also handle here the possibility of repeating the scrolling movement when the user holds down a key for a long period. For doing this, we initiate a `Timer`, called `timer` within the constructor's method of `DisplayItem`. This timer schedules a task and will periodically call to this task. The task is a `TimerTask` object, called `KeepScrollingTask`. When this task is called by the timer, at its turn it will call the `keepScrolling()` method of `DisplayItem` to verify if the item is focused, and if the specific key is still help down, then it will call `scrollTextUp()` or `scrollTextDown()` method, depending on the pressed key. For instant, we fix the period to `SCROLLING_DELAY` = 100 [ms].

3.2.4 Block or Unblock users

This process is entirely handled within the main class `PushToChatMidlet`. The application maintains a list of blocked users, called `blockedUsersList`, which is a `Hashtable`. The table's key is the MAC address of the user, and its value corresponds to the user's nickname.

Normally, a `Vector` could be enough if it was just for keeping a list of MAC addresses. However, when we display the list of currently blocked users during the unblocking process, the nicknames are shown instead of the MAC addresses. It is more convenient for the end-user to read a list of nicknames and chose some of them, than to read a list of the MAC addresses.

We use a `List` called `displayableUsersList` to store and display the list of users. For reminding, a `List` is a `Screen`'s subclass, like `Form`, and represents an entity that can be shown on the device screen. The list's type is `MULTIPLE`, it means several choices can be selected at the same time. From the *Chat environment* state, the user can invoke the `blockUsersCommand` command to block users or invoke the `unblockUsersCommand` command to unblock currently blocked users. When one of these commands is invoked, the listener will call the `commandAction()` method to perform defined actions. We employ the `performingAction` variable, which can be equal to one of two constants `BLOCK_USERS` and `UNBLOCK_USERS`, to store the current user's will. `BLOCK_USERS` indicates that the local user is blocking other users, and `UNBLOCK_USERS` indicates the opposite. Then `performingAction` is passed as argument to `showUsers()` method for displaying the corresponding list of users.

In this `showUsers()` method, depending on the value of `performingAction`, each referred `User` from `usersList` for blocking or from `blockedUsersList` for unblocking will be recovered and its nickname will be employed to create an entry in `displayableUsersList`. For doing this, we call its `append()` method. As the user's nickname are shown on the screen, but we need the MAC address to identify an user, a `Vector` named `macAddInDisplayableUsersList` is used to store the MAC addresses at the same position as the corresponding nickname is stored in `displayableUsersList`. This vector is used to recover the corresponding MAC address when some nickname has been selected from the list. Once the vector and the list are prepared, we execute the `display.setCurrent(displayableUsersList)` instruction to display the list on the device screen.

While `displayableUsersList` is shown, the user can invoke `validateCommand` to validate its choices. Once again, depending on the value of `performingAction`, the corresponding method for blocking (`blockUsers()`) or unblocking (`unblockUsers()`) is called.

In order to retrieve selected elements from the list, we call the following instructions:

```
boolean[] selectedUsers = new boolean[displayableUsersList.size()];
displayableUsersList.getSelectedFlags(selectedUsers);
```

The value of the `n`-th element of the array `selectedUsers` will indicate if the `n`-th element in `displayableUsersList` is selected. For example, if `selectedUsers[0]` is true, it means that the first nickname is selected. This index is also used to retrieve the corresponding MAC address in `macAddInDisplayableUsersList`. If an user is selected for being blocked, its MAC address will be added to `blockedUsersList` and will be removed from `usersList`. Otherwise, if it's selected for being unblocked, its MAC address will simply be removed from `blockedUsersList`. It implies that at the next reception of any payload from this user, it will be treated as a *New user* case inside the message reordering process. Then, we reset the list containing user's nicknames and the vector containing MAC addresses for the next use. Calling `showChattingArea()` method will lead the application to the previous *Chat environment* state.

3.2.5 SLEF status

This process is also handled within the main class `PushToChatMidlet`. In order to display the status of the lower SLEF layer, we use another `Form` called `statusForm`. It is the simplest form in the application. It contains only a command, named `cancelCommand`, for returning to the chat environment, and a `StringItem` for displaying the content, named `slefMonitoringArea`. The content to displayed on the device screen is provided by the `SLEFProtocol` instance via `setMonitoringAreaContent()` method. The content contains information related to the SLEF protocol, like the current number of packets in the epidemic buffer or the attributes of these packets, such as age and the number of sent events. If `cancelCommand` is invoked, `showChattingArea()` method will be called to lead the application to the previous *Chat environment* state.

3.2.6 User's identification

The `User` class represents an user in the network. We need to identify each user with an unique identifier. The simplest idea is to use the nickname as the identifier. However, if two users chose the same nickname, they will be confused in their messages will not be correctly delivered.

A second solution is to use the MAC address of the device. The J2SE class that gives the possibility to retrieve the MAC address of a device is `java.net.NetworkInterface`. But, regretfully, due to security reason, it is not possible to have a low-access on device in J2ME's CLDC. Until now, retrieving MAC address is still a difficult topic to resolve in J2ME community. Another regretful matter is that J2ME's CLDC does not support *Java Native Methods* (JNI). The idea could be to write a method in another language which device's low-access is allowed, for example in C, and then call the method from the J2ME application.

Another idea for the identifier is to use the Smartphone's *International Mobile Equipment Identity* (IMEI) number. This number usually resides in system's properties. In Java, we can query system's properties by using the `java.lang.System.getProperty()` static method, as in:

```
String key = "microedition.platform";
String value = java.lang.System.getProperty(key);
```

In the case of the Smartphone HTC S620, the returned value is "Windows CE 5.1", which indicates current platform of the device. Once again, not all system's properties are supported in CLDC/MIPD 2.0. It depends on the J2ME implementation, and further, on device's manufacturer to develop and provide additional APIs to retrieve the optional properties. Nokia and Sony Ericson are two manufacturers who provide additional APIs built on CLDC for reading the optional information of their phones. The IMEI number can be retrieved on Nokia phones by giving "com.nokia.IMEI" as they key and "com.sonyericsson.imei" for Sony Ericson phones. Unfortunately, HTC does not provide any additional API for their phones.

As a result, this user's identifier still is a problem to be solved. For instant, after the *Ask for nickname* state of the application, we call `retrieveMacAddress()` method in `PushToChatMidlet` class on the purpose of retrieving the MAC address of the device. However, due to the problem explained above, this method returns the value of the user's nickname. Within the development, we still refer to MAC address as the user's identifier in the network, but the contained value is the user's nickname. The idea is not to let further development have to change all the program, but only the returned value of this `retrieveMacAddress()` method.

3.3 Java Standard Edition version

The second version of the application was developed in J2SE for being deployed on desktops or laptops. The provided package `PushToChatSE.Application` contains all necessary classes for this J2SE version. There are only 3 classes in the package, which are `PushToChat`, `Payload` and `User`. The last two classes are identical to the J2ME's ones described in section 3.2. That also is the case for the message reordering process (section 3.2.2) which is handled in `PushToChat` class.

We would like only mention the development of this version. We do not intend to go into its details. The first reason is the project focuses on development in J2ME for Smartphones, and the second reason is this J2SE version has the same structure as the J2ME one. The main difference resides on the way that the graphical user interface was built.

Besides, this version implements all required functionalities and is fully compliant to the J2ME one. It gives the possibility to interconnect different types of devices, from Smartphones to traditional desktops, and to make up an interesting network.

Chapter 4

Self Limiting Epidemic Forwarding (SLEF) Implementation

4.1 Protocol description

Self Limiting Epidemic Forwarding (SLEF) was designed by Alaeddine El Fawal as *"a complete practical middleware for multi-hop broadcast in ad hoc networks. It adapts itself to the variability of the ad hoc network environments"'*[5]. For this end, it uses multiple advanced mechanisms. We are taking the liberty to refer the reader to the main article of SLEF, titled *"Multi-hop Broadcast from Theory to Reality: Practical Design for Ad Hoc Networks"* for understanding SLEF. Therefore, we will only give a brief description about its functions.

1. *Congestion control*: This function consists in adapting the application injection rate for (1) avoiding local buffer overflow and (2) allowing packets to propagate in the network.

2. *Efficient use of MAC broadcast*: This function manages an efficient use of the MAC broadcast by using two mechanisms: pseudo-broadcast and presence indicator.

3. *Scheduler/fairness*: This function is for deciding which packet to serve. It ensures some level of *source-based fairness*.

4. *Buffer management*: This function is responsible of dropping packets in order to keep space in the buffer.

5. *Spread control*: This function manages the trade-off between the spread and the application rate. It uses an `aging` mechanism.

6. *Inhibition*: This function inhibits nodes from transmitting over-sent/received packets. It uses a concept called *virtual rate* to achieve an adaptive inhibition.

4.2 Protocol implementation

The implementation was first realized in J2ME, and then another version in J2SE has been proposed. However, as there isn't any need of the user interface in this layer, the two versions are identical. In consequence, we will only present in this chapter the J2ME implementation.

The provided package `PushToChat.SLEF` contains all necessary classes. The following will describe each of them.

NetMessage.java This represents the unit used to exchange information between SLEF layers. As illustrated in figure 4.1, a `NetMessage` contains a payload, then the source and the sequence number of this payload. Actually, this source corresponds to the MAC layer of the originator's device, and the `NetMessage` sequence number is retrieved from the payload's one. The field `age` that a `NetMessage` carries is the attribute *age* of a packet within the SLEF protocol[5]. And the last field `sender` indicates the originator of the `NetMessage`. It is used to verify in the application layer weather the received payload has been transmitted by its originator, or it has been forwarded by some other node in the network. This process is not included in the protocol, but its objective is to demonstrate one of the advantages of SLEF. Another utility of this `sender` field is for discarding at the reception the packets transmitted by the local node. Indeed, while using the broadcast address to transmit datagrams with an UDP connection, it happens that these datagrams are also received and delivered by the local node. This will make the SLEF's operation error erroneous. To resolve this problem, while reconstituting the `NetMessage` delivered by the MAC layer, we verify weather this `NetMessage` was transmitted by the local node, by checking the `sender` with the local MAC address. If it is the case, this `NetMessage` will be discarded.

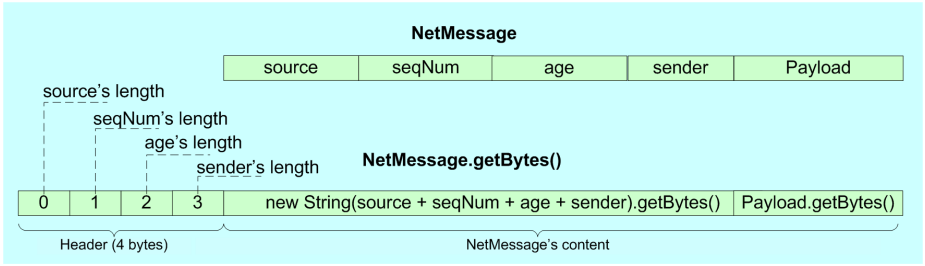


Figure 4.1: NetMessage

As it is the case in `Payload` class, `NetMessage` also provides a method for converting the current state of the object into an array of bytes, which is called `getBytes()` method. Indeed, this mechanism is not used in the application layer, where the `Payload` class is defined. As the `Payload` is encapsulated in this SLEF layer, its `getBytes()` method also is called in this layer.

The `bytes` array of `NetMessage` is composed of a header part and a main content part. The header has 4 `bytes`, each of them indicates the length of its corresponding

field in the content part. The length of a field is the number of character it has. This length is an integer converted into a `byte`. As an integer is a 32-bit, then only its eight least significant bits will be taken into account. Therefore, the maximal length for a field is 127, which can be considered more than enough. As an example, if the value contained in the first `byte` (labeled *source's length*) corresponds to an integer 5, it means that the first five bytes of the content compose the `source` field. Then, the first part of the content contains the bytes of all fields concatenated into a single `String`, and the second part is the content of the `Payload`. For copying an array to another, we user the static method `System.arraycopy()`. By processing in the opposite direction, we can reconstitute the `NetMessage` whose the bytes array is delivered by the MAC layer.

NetRecord.java This class corresponds to the `clone` notion in SLEF protocol. It contains the original packet, which is a `NetMessage` and its attributes: the number of sending events `sendCount`, or of receiving events `rcvCount`, the virtual rate `vRate` and `age`. These are ones of most essential attributes of the protocol. We also define another variable called `firstSeenTime`, which records the time (in millisecond) that the packet of this clone is received by this node for the first time. This record is used for computing the *real time* age of the packet. Another time's record variable called `lastTransmissionTime` is also present. It records the time at which the packet was transmitted for the last time. This value is used in the decision if the last transmission in pending mode will be taken into account, or this transmission has occurred too long ago for being confirmed. Furthermore, this class also provides methods for computing and updating the attributes mentioned above.

NetRecordId.java This class provides an identification for a `NetRecord` object. It is composed of two global variables `source` and `seqNum`, which corresponds to the MAC address of a node and the sequence number of the payload. With this information, we can recover the corresponding `NetRecord` object. The class only overwrites the `equals()` of the `Object` class for the comparison of two `NetRecordId` objects. We would like to know if those two objects are equivalent, and are only interested in the values of their two fields `source` and `seqNum`. If these values are equal, then we confirm the equivalence of those two objects.

NetSource.java This class represents a source in the network. In a simple way, we can say that this class is to the SLEF layer like the `User` class is to the application layer. This class contains the table of clones called `cloneTable`, which is a `Hashtable`. The table's keys are the packet's sequence numbers, and the values are the corresponding `NetRecord` objects. Then, the class has a `Vector` called `FIFO`, which contains *eligible packets* [5]. Finally, a `double` named `sourceClaim` indicates how much this source can claim to be scheduled is also present.

SLEFTimeoutTask.java This class extends the Java's `TimerTask` and represents the task which will be scheduled by the `SLEFProtocol`'s global timer for the packets stored in the epidemic buffer. The packet's identifier is stored in the `NetRecordId`'s instance, called `recordId`. Therefore, when the timer's delay expires, this task will call the `timeout()` method of the `SLEFProtocol` class and gives the packet's identifier

as argument. This signals that the packet needs to be added into the `FIFO` of the corresponding source. This relates to the *eligible packet*'s concept.

SLEFProtocol.java This is the main class of the SLEF package. It orchestrates the whole operation of the SLEF layer. Among others, we define in this class all SLEF's constants (such as `K0, K1` or `R0`) and the essential epidemic buffer, which will be discussed in section 4.2.1. We would like to present some important components that are used in this class before entering into its main functions.

`blacklist` (Vector): This buffer stores the list of removed packets. These packets have once been present in the epidemic buffer, but are now removed due to its age, to the buffer size or due to the congestion control. In any case, the process keeps tracks of these packets, by adding its corresponding `NetRecordId` object to this buffer, in order for it to be discarded at next reception.

`timer` (Timer): This is the global timer that will be used to schedule the specific task for all packets in the epidemic buffer. It needs two parameters: the task to be executed (`SLEFTimeoutTask`) and the delay after which the execution will start.

`timerTaskTable` (Hashtable): This table maintains a list of packets for which a timer task is currently associated (i.e. scheduled). The table's key is a `NetRecordId` object, which identifies a `NetRecord`, and its key is a `SLEFTimeoutTask` object, which corresponds to the scheduled task . Whenever a timer is set for a packet, an entry will be added into this buffer. This buffer is used for recovering the timer task if it needs to be canceled before its execution after the specific period.

`selfRecordsDroppingList` (Vector): According to the *Congestion control* function, the number of self-packets stored in the epidemic buffer cannot exceed σ (currently equal to 2). But in case of an *implicit acknowledgement* of a given self-packet or in case where its `sendCount` reached 3, the application is allowed to inject a new payload, and the given old self-packet will be removed. We call that old self-packet *removable* one. Hence, this buffer is used to store a list of removable self-packets. The vector only stores the sequence numbers, because we know that the source of these packets is the local node.

`pendingRecordsList` (Vector): SLEF's *Efficient use of MAC broadcast* function discusses about the pending transmission. This buffer is used to store packets whose the last transmission is in this pending mode. The buffer contains the `NetRecordId` objects, from which the corresponding `NetRecord` objects can be recovered.

`localSource` (NetSource) This object represents the local source at this SLEF layer. Its `cloneTable` contains locally originated packets, that we call self-packets. This object can also be recovered from the epidemic buffer (presented later), but because of its frequent use, it's more convenient to maintain it apart.

`sourceClaimIncrement` (double) This is the value by which the `sourceClaim` of each source in the epidemic buffer will be increased whenever a packet is scheduled for transmission (cf. *Scheduler* function). It's equal to $\frac{1}{\#sources}$ and is

updated by a call to `updateSourceClaimIncrement()` method each time a new source is created.

`lastReceiveTime` (long): This variable is used to store the time of the last reception. This value will be employed to compute the `presenceIndicator`, which belongs to *Efficient use of MAC broadcast* function

This main class also contains a instance of the upper layer `PushToChatMidlet` that we call `parent`, and one of its lower layer `MACBroadcast` called `macBroadcast`. These two objects allow the communications between the three layers. We would like to present next the buffer epidemic, and then the implementation will be explained though the two main processes: ***Reception*** and ***Transmission***.

4.2.1 The epidemic buffer

The protocol defines that each node in the network maintains one epidemic buffer to store "received and locally originated packets". In the implementation, the role of this buffer is ensured by a `Hashtable` called `epidemicBuffer`. This table and its connections are illustrated in figure 4.2.

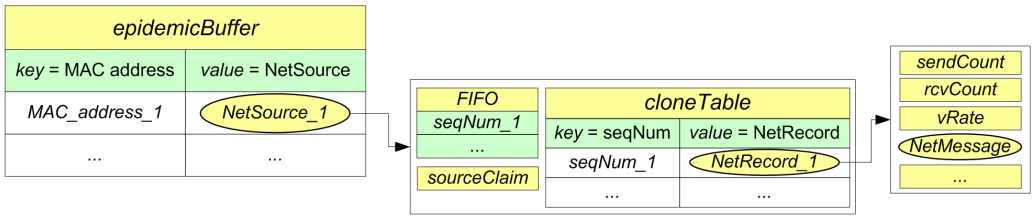


Figure 4.2: The Epidemic buffer

Actually, the buffer does not directly contain the packets. The `epidemicBuffer`'s key is the MAC address of the node and its value points to a `NetSource` object. As mentioned, this latter represents a source in the network. At its turn, each source keeps with it a buffer called `cloneTable`, which also is a `Hashtable`, to store packets originated by itself. The packet here is a `NetRecord` object which is identified by the sequence number. It is necessary to mention that the sequence number of the packet here is the sequence number of the payload created in the application. A packet in SLEF layer always encapsultates a payload. As a result, if we want to verify if a packet exits in the epidemic buffer, we have to recover first its source and then to look for that packet in the clone's table.

When the application starts, an entry for the local user called `localSource` will be created and added into the epidemic buffer. Naturally, this buffer still remains empty because the source's `cloneTable` does not contain any entry.

In [5], it says that the epidemic buffer size is upper bounded by $\frac{maxTTL+1}{K_1}$. By using default values, `maxTTL`=255 and K_1=0.1, we obtain 2560 [packet] for the maximal size of the packet. However, due to the very limited memory space of the Smartphone (64MB of RAM), we fix this limit to 170 [packet].

4.2.2 Reception

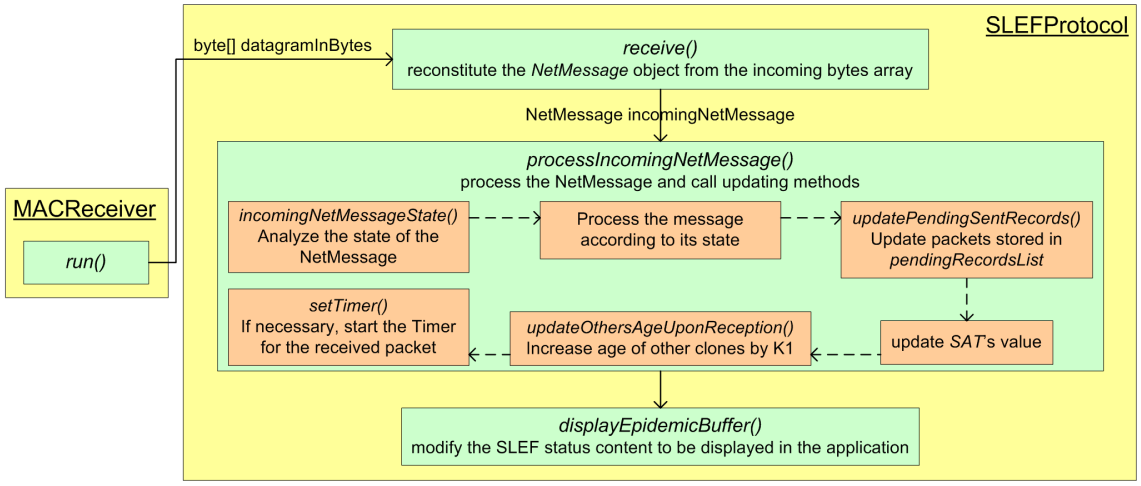


Figure 4.3: SLEF Reception

NetMessage's reconstitution

The reception process is presented in figure 4.3. It begins when the MAC layer delivers the packet to SLEF. That means, when the receiver thread calls `SLEFProtocol`'s `receive()` method with the bytes array `datagramInBytes` as argument. The first action to take is to reconstitute the `NetMessage` object from that array. Basing on the structure defined in `NetMessage` class (illustrated in figure 4.1), we read the first four bytes in order to obtain the length of each field. Afterwards, by using the `String.substring()` method, we can extract each field. Below is an example for extracting the `sender` field of a `NetMessage` object.

```
senderLength = (int) datagramInBytes[3];
datagramContent = new String(datagramInBytes,0,datagramInBytes.length);
sender = datagramContent.substring(4 + sourceLength + identifierLength + ageLength,
       netMessageStartPos + sourceLength + identifierLength + ageLength + senderLength);
```

Following this mechanism, we extract field by field. We reconstitute first the `Payload` object, as it is required to create a `NetMessage` object. Once this latter object is reconstituted, before calling `processIncomingNetMessage()` method to process this message, we record this time in `lastReceiveTime`, which indicate the time of the last reception in millisecond.

NetMessage's processing

The main part of this reception process is realized within this `processIncomingNetMessage()` method. First, we analyze the state of this packet following to the schema in figure 4.4. The `incomingNetMessageState()` method performs this analysis and returns an integer indicating the state. We notice that there are 7 states, nevertheless, only with 3 of them the packet is valid and will be processed. These 3 states are 1, 4 and 7. While with other states, either the `blacklist` contains the packet's identifier, or the packet's age exceeds the

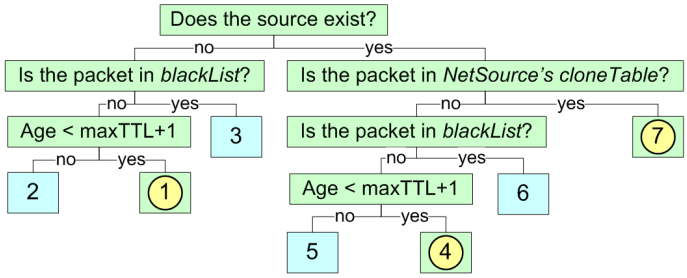


Figure 4.4: Received NetMessage state

authorized one, in consequence it will be discarded. If it's because of the age, this packet will also be added into the `blacklist`. We will now discuss about the three valid states.

State 1: This is the first packet that we receive for that source. A new `NetSource` object and a `NetRecord` one will be created. Before inserting the clone into the source's clone's table, we have to update its received events counter `rcvCount`, its virtual rate `vRate`, and its age. The `NetRecord`'s `receiveCloneUpdate()` method performs this update and is shown below.

```
public void receiveCloneUpdate() {
  rcvCount++;
  vRate = computeVRate();
  // update age
  discreteAge += SLEFProtocol.K0;
  continuousAge = computeContinuousAge();
  age = discreteAge + continuousAge;
}
private double computeContinuousAge() {
  return (((double)((new Date().getTime()) - firstSeenTime))/3600000.0)
                                        * SLEFProtocol.alpha; // [hours]*alpha
}
```

Inside this method, the `computeVRate()` method applies the formula $R_0 a^{rcvCount} b^{sendCount}$ and returns a double. While `computeContinuousAge()` computes the continuous age of the packet. The first subtraction `(new Date().getTime()) - firstSeenTime` gives us the elapsed time in millisecond since the moment this packet was received for the first time. Then, dividing it by 3600000 converts this time to hour, which will be multiplied by the constant α of SLEF to obtain the continuous age of the packet. This constant α is equal to 32, which will let the packet reside in the epidemic buffer during at most 8 hours.

Afterwards, this packet is ready to be inserted into the epidemic buffer. This insertion increases the buffer size. If the maximal size is reached, the process will remove the oldest packet in the epidemic buffer. This follows the rule of *Buffer management* function. The system maintains a global variable called `oldestRecordId`, representing the packet with highest age in the epidemic buffer. When this packet needs to be removed, we call `updateOldestRecordId()` method to update this variable, by

searching again in the epidemic buffer the packet with the highest age, and then we call `removeOldestRecord()` to remove it. As we have just added a new source, the value of `sourceClaimIncrement` also needs to be updated. And this is the first packet received for this clone, the corresponding payload is also passed up to the application layer, by a call to `parent.processIncomingMessage()`. This implies that the process only delivers the same payload once to the application layer. Otherwise, the *message reordering process* would not correctly operate.

State 4: This state is quite similar to the previous case, except that the source already exists in the epidemic buffer. Hence, we will recover this source and create a new clone, perform reception's updates and add this packet into the buffer. The corresponding payload will also be delivered to the application layer, as this is the first reception.

State 7: In this case, both the source and the clone are already registered. Hence, we will recover them both, and call `receiveCloneUpdate()` method to update the packet's attributes. Moreover, if the packet is a self-packet (i.e. the MAC address it carries is equal to the local one), then we will consider this reception as an implicit acknowledgment, according to the second statement of SLEF's *Congestion control* function. Then, this self-packet is a *removable* one and therefore will be added into `selfRecordsDroppingList`. The presence of a removable self-packet implies that SLEF layer allows the application to inject new packets. The call to `enableSending()` of the `PushToChatMidlet` class will set its variable `allowedToSend` to *true*.

Timer

Up to this moment, the state of the received `NetMessage` is identified. If the packet is a valid one (state equal to 1, 4 or 7), then we need to set a timer task for it. However, if the state is equal to 7 and the `pendingRecordsList` contains this packet's identifier, then this operation will be postponed to the next part of the process.

To set a timer for a packet, we'll call `setTimer()` method. It is possible that there's another timer task which is currently scheduled to this packet, but thes specified delay does not expire yet. Hence, within the `setTimer()` method, before associating a new task for this clone, a call to `cancelTimer()` method will first try to cancel the current one, if it does exist. The corresponding `SLEFTimeoutTask` object will be recovered and its `cancel()` method is excuted. It is good to mention that once a `TimerTask` instance has been cancelled, it is not possible to use this task again for a future scheduling. Otherwise, it would be preferable to reschedule this instance again for the packet, and we would have avoided the destruction and construction operations.

This timer process corresponds to the notion of *"eligible"* packets within the *Scheduler* function. It says that a packet is eligible if at any time `t`, it has `earliestSendTime` $\leq$ `t`, where `earliestSendTime` is equal to the last time the `vRate` was modified, plus $\frac{1}{vRate}$. It also means that, at the moment where its `vRate` is modified, if the packet waits for a period equal to $\frac{1}{vRate}$, then it will become eligible. And that's how we compute the delay for the timer, which is equal to $\frac{1}{vRate}$. A timer is necessary for a packet to become eligible, thus to be inserted into the source's FIFO. The actions to take when a timer's delay expires (called *timeout*) will be discussed later in the transmission/retransmission mechanism (section **??**).

Pending transmission's update

The next action to take is to confirm the transmission of packets that were previously transmitted in pending sending mode. This confirmation will occur if the `pendingSendConfirmation` flag is *true* and the `pendingRecordsList` is not empty. If all conditions are verified, then the method `updatePendingSentRecords()` will be called. The condition for a pending transmission to be confirmed is that the elapsed time from the moment where the transmission occurred to now must be less or equal to a predefined interval, called `validPendingInterval`, which is currently fixed to 1000 millisecond.

For each packet's identifier stored in the `pendingRecordsList`, the corresponding packet (NetRecord) will be recovered. Then, the lastTransmissionTime field of the packet will be used to verify the condition above. If the pending transmission has not occurred too long ago, it will be updated as if it was a normal transmission. This update is ensured by the `NetRecord`'s `sendCloneUpdate()` method, which is very similar to `receiveCloneUpdate()` that we discussed earlier. The sole difference is that instead of the `rcvCount`, its `sendCount` will be increased and hence also be the virtual rate. And as the virtual rate has changed, we need to set a new timer for this packet. The action is just discussed above. The timer for the received packet whose the state is equal to 7 and which is present in the `pendingRecordsList` is performed here.

```
if ((currentTime - record.lastTransmissionTime) <= validPendingInterval) {
  record.sendCloneUpdate();
  setTimer(recordId);
  if ((recordId.source.equals(this.localMacAddress)) && (record.sendCount==3)) {
    if (!selfRecordsDroppingList.contains(seqNumInInteger)) {
      selfRecordsDroppingList.addElement(seqNumInInteger);
    }
    parent.enableSending();
  }
}
```

Moreover, if the packet is a self-packet and the new value of the `sendCount` is greater or equal to 3, then the third statement of the *Congestion control* function will be applied. The result will be the same as *State 7* above, this packet will become a removable self-packet and will be stored in the `selfRecordsDroppingList` buffer. Finally, `PushToChatMidlet`'s `enableSending()` is called.

However, if the pending transmission has occurred too long ago, we will not confirm this transmission and will add right after the packet's identifier into the source's `FIFO`, if this latter does not contain the packet's identifier yet. This action aims to let the packet to be retransmitted as soon as possible. Before doing this, if a timer task is currently scheduled for this packet, it has to be canceled first.

```
else if (!source.FIFO.contains(seqNumInInteger)) {
  cancelTimer(recordId); // cancel the current timer if necessary
  source.FIFO.addElement(seqNumInInteger);
}
```

At the end of this update, the packet will be removed from `pendingRecordsList` buffer. This process continues until the buffer becomes empty.

SAT and reception's updates

The received `NetMessage`'s processing is done, now other updating methods are necessary. First, we will update the value of `SAT`, which is the *Self Age Threshold* value. The new value for this threshold is equal to $max(SAT_0, SAT - K_1)$ (with SAT_0= 10).

Afterwards, we have to increase the age of other packets in the epidemic buffer by K_1 (*adaptive age*). The method `updateOthersAgeUponReception()` is responsible of this action. It recovers each packet in the epidemic buffer, and then distinguishes two cases: self-packet and foreign-packet. For the latter one, the process is simple. The `NetRecord`'s `updateUponReceptionForForeignPacket()` is called for each packet. This increases the discrete age by K_1 and the continuous age is recomputed with `computeContinuousAge()` that we discussed earlier. Then, the new age is the sum of these two. If the new age exceeds `maxTTL+1`, then the packet is too old and will be dropped. We also take advantage of this process to update the variable `oldestRecordId`, which represents the packet with highest age in the buffer.

For self-packets, we applies in this situation the *density detection mechanism*, for short, we call it *"SAT process"*. We will call first the `updateUponReceptionForSelfPacket()` method in order to update the self-packet's age. This method receives in parameter the current value of `SAT`. It will compare the currnet age of the packet with this `SAT` value. If the current age reaches this value and the packet was never transmitted (`sendCount` is equal to 0), then its age will be equal to `maxTTL`. Otherwise, the self-packet's age will be updated as it is in the foreign-packet case. After its age is updated, if the packet is too old (greater than `maxTTL+1`), it will be removed from the epidemic buffer. Otherwise, it will be taken in account for the update of the variable `oldestRecordId`. We would like to mention that it is not necessary to verify here whether the self-packet's `sendCount` is strickly positive before removing it. Because even if the *SAT process* is activated for this self-packect, its age will never exceed `maxTTL+1`.

SLEF status displaying

The last call we perform with this process is to the `displayEpidemicBuffer()` method. This method browses the epidemic buffer and generates a `String` called `textContent` containing the current state of the packets (i.e. its attributes). Then this `textContent` is passed up to the application layer. The content can be displayed right after, as it is the case in the J2SE application, or by the user's command, as in the J2ME version.

4.2.3 Transmission/Retransmission

In this section, we will discuss about the sending process of the SLEF layer. We slightly distinguish the transmission process from the retransmission one. The transmission occurs when the application injects a new payload into the SLEF layer and is sure the SLEF will do its best to send this payload to the network, and the other one is due to the retransmission mechanism of SLEF regarding to packets in the epidemic buffer. There are four essential methods involved in this transmission/retransmission: `send()`, `timeout()`, `scheduler()` and `sendToMac()`. The first method, `send()` is a communication point with the application layer. It is called when this latter has a new payload to send over the network. Hence, we

say that this method is responsible for the transmission and its operation will be discussed in the first part of this section. The second method timeout() is called when the delay of a packet's timer expires. This packet becomes eligible and therefore is inserted into the FIFO for a retransmission. In consequence, this method is involved in the retransmission process. Then `scheduler()` is responsible for the *Scheduler* function which selects the next packet to be retransmitted and `sendToMac()` this method is used for delivering the packet to the lower MAC layer. The last three functionned will be discussed in the second part of this section, the retransmission part.

Sending packets coming from the application layer

This process can be divided into two parts. The first part is responsible of the *Congestion control* function and the second part consists in adding the packet into the epidemic buffer and in transmitting it to the MAC layer.

The injection rate of the application is controlled by the SLEF layer. This control is performed through the boolean `allowedToSend` at the application side. The value of this variable is changed by the call to `PushToChatMidlet`'s `enableSending()` or to its `disableSending()` method by SLEF. When `allowedToSend` is set to true, the application can inject new payload into the SLEF layer. For this mean, the application will call the `SLEFProtocol`'s `send()` method, and passes the new payload in paramater.

As mentioned earlier, the number of self-packets currently stored in the epidemic buffer cannot exceed σ, which currently is equal to 2. The `localSource.cloneTable.size()` method returns this number. Therefore, if it is smaller than $\sigma - 1$, the application can freely send new payloads without any restriction from SLEF. The two relevant cases are when it reaches $\sigma - 1$ and σ.

Case $\sigma - 1$: The current number of self-packets is $\sigma - 1$, that means, with the new payload, it will be equal to σ. In a simple way, SLEF would not allow the application to inject any new payload until one of the currently stored is removed. However, according to three possibilities for a new payload be injected defined in *Congestion control* function, we still allow the application to send new payload only if there is any removable self-packet in the epidemic buffer. That means, `selfRecordsDroppingList` is not empty. Otherwise, SLEF will prohibit the application from sending by calling the `parent.disableSending()` method. This call will set `allowedToSend` to false.

Case σ: This case occurs only when at the previous case, the `selfRecordsDroppingList` buffer is not empty. The process will recover the first element in this buffer and then remove the corresponding packet from the epidemic buffer. Afterwards, if there isn't any other removable self-packet, the process will prohibit the application from sending. Otherwise, the application still is allowed to send until the `selfRecordsDroppingList` buffer becomes empty.

```
Integer seqNum = (Integer) selfRecordsDroppingList.firstElement();
NetRecord removeRec = (NetRecord)localSource.cloneTable.get(seqNum);
removeRecord(new NetRecordId(removeRec));
```

```
if (selfRecordsDroppingList.size() == 0) {
  parent.disableSending();
}
```

The congestion control process ends here. We will now add the packet into the epidemic buffer. For this mean, a new `NetMessage` object encapsulating the payload is created, followed by a `NetRecord` object. This new clone will be first into the source's `cloneTable`. As this packet is about to be sent, it's also inserted into the source's `FIFO`. Then, we verify the epidemic buffer size, as a new entry was just added. If necessary, the oldest packet will be removed.

At this state, this new payload is not processed yet by the application's `message reordering process`, hence either is displayed on device screen. We would like to be sure that the packet resides in the clone's table before displaying it at the application side. We deliver back this payload by the call to the `parent.processIncomingMessage()` method.

At any time, the MAC layer can be occupied due to the retransmission mechanism. We maintains a boolean named `MacIdle` to indicate whether the MAC layer is idle. If it is idle, then new packet can be sent to the MAC layer right away. The `sendToMac()` method is responsible for this action and will be discussed in the retransmission part. However, before that method is called, we set `MacIdle` to false, to indicate to other methods that the MAC layer is occupied. This variable can be considered as *mutex* for sharing access to the `sendToMac()` method. In the case where the channel is occupied, the packet cannot be sent right away. Anyways, it resides in the source's FIFO, depending on the value `sourceClaim`, the packet will be scheduled for transmission later.

Retransmission of packets in the epidemic buffer

Timeout

When the delay of specific task `SLEFTimeoutTask` scheduled by the global timer expires, the task will be executed. The only instruction that is defined in its `run()` method is the call to the `timeout()` method of the `SLEFProtocol` class, by giving in parameter the `NetRecordId` object identifying the packet to which the timer is associated. The `SLEFProtocol`'s `timeout()` method can be considered as a retransmission trigger, as its task is too insert the eligible packet into the source's FIFO. When this method is called, the source of the packet will be recovered, then its clone and its sequence number.

```
NetSource source = (NetSource) epidemicBuffer.get(recordId.source); // source
Integer seqNumInInteger = new Integer(recordId.seqNum);             // sequence number
NetRecord record = (NetRecord)source.cloneTable.get(seqNumInInteger); // the clone
```

Then, the method adds the packet's sequence number into the source's FIFO for retransmission. If the channel is idle, it will create a new `NetMessage` object, equal to the current one stored in the clone, but with the current age. Otherwise, as the same as the `send()` case, the packet resides in the source's FIFO and will be scheduled for retransmission by the scheduler.

```
source.FIFO.addElement(seqNumInInteger); // add the packet to its source's FIFO
if (MacIdle) {                           // send to MAC layer
  NetMessage message = record.netMessage;
  message.age = record.age;
  MacIdle = false;
  sendToMac(message);
}
```

Scheduler

In the implementation, the `scheduler()` method is called after a packet has been delivered to the MAC layer, that means, at the end of the `sendToMac()` method. Its tasks are well defined: update the `sourceClaim` for each source, then decide the next packet to serve.

Actually, these two tasks are delegated to the `updateSourceClaim()` method. The scheduler calls this method and waits for the selected packet returned by this call.

```
NetRecordId nextSendRecordId = updateSourceClaim();
```

The `updateSourceClaim()` method recovers each source stored in the epidemic buffer and increases its `sourceClaim` by the `sourceClaimIncrement`. Then it compares the new value `sourceClaim` of the source with currently highest one, stored in `maxClaimSource`. If `sourceClaim` is greater than `maxClaimSource` and the source's FIFO is not empty, i.e. there exists eligible packets, then the source's key will be stored into `selectedSource`.

```
while (sourceKeys.hasMoreElements()) {
  currentKey = (String) sourceKeys.nextElement();
  currentSource = (NetSource) epidemicBuffer.get(currentKey);
  if (currentSource != null) {
    currentSource.sourceClaim += sourceClaimIncrement;
    // select the source with the highest sourceClaim
    if ((maxClaimSource < currentSource.sourceClaim) && (currentSource.FIFO.size() > 0)) {
        maxClaimSource = currentSource.sourceClaim;
        selectedSource = currentKey;
    }
  }
}
```

At the end of the process, the method will create a `NetRecordId` object corresponding to the first packet stored in the FIFO of the selected source `selectedSource`. This object is returned to the scheduler and is stored in `nextSendRecordId`. If the returned object is not null, then the scheduler will create a `NetMessage` object from `nextSendRecordId`, and update its age before passing it to `sendToMac()`. At this state, the channel is occupied (`MacIdle` = false). Therefore, the scheduler will keep the channel until no eligible packet is present in the epidemic buffer anymore. In this case, the scheduler releases the channel by setting `MacIdle` to false.

```
if (nextSendRecordId != null) {
  Integer seqNumInInteger = new Integer(nextSendRecordId.seqNum);
  NetSource source = (NetSource) epidemicBuffer.get(nextSendRecordId.source);
```

```
  NetRecord record = (NetRecord) source.cloneTable.get(seqNumInInteger);
  NetMessage message = record.netMessage;
  message.age = record.age;
  sendToMac(message);
}
else {
  MacIdle = true;
}
```

Deliver the packet to MAC layer

As mentioned above, the `sendToMac()` method delivers the packet (`NetMessage`) to the MAC layer and performs updates depending on the value of the `presenceIndicator`. The presence indicator is a concept defined in *Careful use of MAC broadcast* function. This method can be called from the scheduler, from `timeout()` or from `send()` methods. The two latter cases happen will the channel is idle, i.e. `MacIdle` is true.

Before sending the `NetMessage`, this method recovers its source and decreases the value of `sourceClaim` by 1, as defined in *Scheduler* function. Then, this packet is also removed from the source's FIFO, as the packet will be transmitted soon.

```
source.sourceClaim--; // decrease the claimSource by 1
Integer seqNumInInteger = new Integer(netMessage.identifier);
if (source.FIFO.contains(seqNumInInteger)) {
  source.FIFO.removeElement(seqNumInInteger);
}
```

Next, the method will put the MAC address of the local source as in the `NetMessage`'s `sender` field. For reminding, this field indicates the node that transmits this packet. Then it calls the `send()` method of `MACBroadcast` class and gives the bytes array of the `NetMessage` as parameter. This `send()` method is a blocking one. It returns a boolean, which is true when the MAC layer has successfully delivered the packet into the network, or is false when an error or an exception has occurred. We keep sending this packet if the transmission is not successful.

```
netMessage.sender = parent.getMacAddress(); // Modify the sender field of the NetMessage
while(!macBroadcast.send(netMessage.getBytes())); // Send to MAC layer
// compute the presence indicator
presenceIndicator = (new Date().getTime() - this.lastReceiveTime) < this.busyInterval;
```

When the transmission is successful, then we will compute the value of `presenceIndicator`. It will depend on the amount of elapsed time since the last reception. We remind that the value of `lastReceiveTime` is updated at each reception. If this amount of time does not exceed `busyInterval`'s value, which is arbitrarily fixed to 5 seconds, then `presenceIndicator` is set to *true*, otherwise, it is set to *false*. The `busyInterval`'s value should not be chosen in an arbitrary manner, but it should be based on the frequency of interactions between nodes. However, this value is not analyzed in this project.

With the given value of `presenceIndicator`, we can decide whether the transmission was in pending mode. If `presenceIndicator` is *true*, then the transmission was sent in normal

mode and can be considered successful. In this case, we update the `sendCount` and other attributes of the packet with the `sendCloneUpdate()` method, and set `pendingSendConfirmation` to *false*. This method was once mentioned during the reception process. It increases the `sendCount` by 1, re-computes the `vRate` and `age` of the packet.

As its age increases, we verify that it does not exceed `maxTTL` +1, otherwise, it will be removed from the epidemic buffer. We call `removeRecord()` for removing the packet. Else, if this packet does not age out yet, as its `vRate` has been updated, the timer will schedule a new timer task for this packet. `setTimer()` is responsible for this action.

Afterwards, if the packet is a self-packet and the new value of the `sendCount` is greater or equal to 3, then we apply once again the third statement of the *Congestion control* function. This packet is a removable self-packet and is added in the `selfRecordsDroppingList` buffer. Finally, we allow the application to send new packet by a call to its `enableSending()` method.

These actions are taken when the transmission is not in pending mode. In the contrary case, we do not increase the packet's `sendCount`, but only its continuous age is increased. Also, the `pendingSendConfirmation` is set to *true* to indicate that the system is in pending sending mode. As this transmission can be analyzed later for being validated, we have to record the time of the packet's transmission to its variable `lastTransmissionTime`.

```
pendingSendConfirmation = true;
record.updateContinuousAge();
record.setLastTransmissionTime(new Date().getTime());
NetRecordId recordID = new NetRecordId(record); // the identifier of the record
if (!pendingRecordsList.contains(recordID)) {
  pendingRecordsList.addElement(recordID);
}
setTimerInPendingMode(recordID); // set a new timer for this packet, but in sending pending mode.
```

Then we add this packet into `pendingRecordsList` buffer and associate a timer to this packet by the `setTimerInPendingMode()` method. The only difference between this method and `setTimer()` resides on the timer's delay. In pending mode, the packet's `vRate` is maintained and we use a fixed delay for its timer, called `pendingRetransmissionDelay` and currently fixed to 5000 milliseconds.

The value of `SAT` is also updated in this process, which is equal to $min(maxTTL, SAT + SAT_0)$. As there's change within the epidemic buffer, `displayEpidemicBuffer()` is called to redisplay the status and finally the scheduler will take place in order to select next eligible packet for retransmission.

Packet's removal

Packets can be removed from the epidemic buffer according to the *Buffer management* and the *Congestion control* functions. The `removeRecord()` method is responsible for this removal. It has to ensure that the packet is completely removed and does not let any trace that could disturb SLEF's operation. That is to say, the timer must be canceled (using cancelTimer() method), the clone must be removed from the source's clone table

and also from the source's FIFO. After that, the packet is added into the `blackList` for not being processed again. With that, the epidemic buffer size can be decreased by 1. Finally, if a self-packet is removed, we verify the number of self-packet currently stored in the epidemic buffer for allowing the application to send new packet, due to the *Congestion control* function.

4.2.4 Checklist

At this state, almost all SLEF's functions are implemented. The only missing mechanism is the use of the pseudo-broadcast, which is included in the *Careful use of MAC broadcast* function. We would like to give a brief checklist basing on the SLEF's functions.

1. *Congestion control*: This function is first applied when a new payload is injected by the application through the `send()` method. Then, it's also applied when a packet is received, which is handled by the `processIncomingNetMessage()` method. Finally, we verify this function when a packet is removed from the epidemic buffer with the `removeRecord()` method.

2. *Efficient use of MAC broadcast* The pending sending mode is supported and the presence indicator is introduced.

3. *Scheduler* The `scheduler()` method ensures this function. It selects eligible packets in accordance with the source-based fairness. For this mean, we employ a global timer to schedule a new timer task for the packet each time its `vRate` changes, and the `sourceClaim` of each source is updated at each transmission.

4. *Buffer management* We fix a maximal size `maxEpidemicBufferSize` for the epidemic buffer and always verify its size whenever a new packet is inserted. We keep track on the packet with the highest age for a removal if necessary.

5. *Spread control* Age of each packet is always re-computed upon a reception or a transmission. Even in the pending sending mode, the real time age is always updated. We also implemented the *SAT process* for self-packets.

6. *Inhibition* The virtual rate of each packet is well maintained within the `NetRecord` class. All necessary methods for updating are implemented.

Chapter 5

MAC Broadcast

We aim to present in this chapter the design and the implementation of the third and also the last layer of the architecture, the MAC layer. The role of this layer is very simple, it is responsible for broadcasting the packets into the network and for receiving them from the network. Therefore, its design is only composed of two classes: `MACBroadcast` for broadcasting and `MACReceiver` for receiving. Because of the simplicity of the design, it can be used for both J2ME and J2SE versions. An illustration is given in figure 5.1. We will give here the main structure for both of them, then the particularity of each version will be discussed in their implementation part.

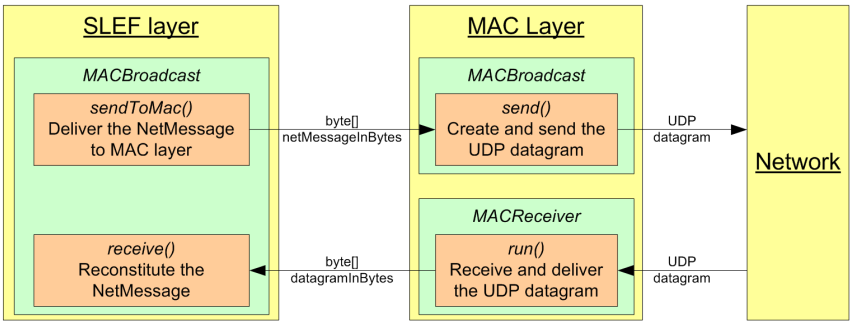


Figure 5.1: MAC layer's design

The unit used for exchanging information between the SLEF layer and this layer is an array of bytes. And, as we would like to keep this layer as simple as possible, there will not be any data processing in this part.

The `MACBroadcast` is the main class of this layer. It is responsible for opening the connection, which is an UDP connection, and then for initializing an instance of `MACReceiver`. Along with the constructor method, we define in this class the `send()` method, which is used by the SLEF layer to deliver packets, which are bytes arrays. Then this method is supposed to create an UDP datagram from these arrays and broadcast them into the network. It will return a boolean set to true if the transmission was successful, and it is set to false if errors occured.

Besides, the `MACReceiver` extends `Thread` and contains only the the constructor and the run() methods. It maintains a `MACBroadcast` object and a `SLEFProtocol` object. The first one is used to retrieve the connection for sharing, and the second object is used to call the corresponding method to deliver incoming datagram to.

5.1 J2ME implementation

5.1.1 MACBroadcast

Within the constructor method of `MACBroadcast`, we open an UDP connection called `datagramConn`.

```
datagramConn = (UDPDatagramConnection)Connector.open("datagram://:" + portNumber);
```

The IP address is not given while opening the connection because we would like to obtain a datagram server. The variable `portNumber` indicates the port number and is equal to 9001, a standard port for UDP connection. In J2ME, `UDPDatagramConnection` is a subclass of `DatagramConnection`. The only difference resides on the fact that `UDPDatagramConnection` provides an additional method called `getLocalAddress()` which returns the IP address of the device. Then, a `MACBroadcast` instance is created and started within the `init()` method. SLEF layer only calls this method when at the application layer, the user has correctly inserted his nickname. It is for avoiding the messages to be receieved while the user is not in the chat environment yet.

When SLEF wants to deliver a packet,it will call the `send()` method. An UDP datagram will be created and sent with the predefined broadcast address, called `broadcastAddress`.

```
datagramConn.send(datagramConn.newDatagram(netMessageInBytes, netMessageInBytes.length,
                                                                broadcastAddress));
```

This the value of `broadcastAddress` is "datagram://169.254.255.255:9001" when the Smartphones HTC S620 are used. This value was deduced from the register key `AutoSubnet` of the Smartphone, which is equal to "169.254.0.0" and can be found in `HKEY_LOCAL_MACHINE\Comm\TNETWLN1\Parms\TCPIP\`. This is because in J2ME, due to security reason, the generic broadcast address 255.255.255.255 is not allowed.

5.1.2 MACReceiver

This thread keeps receiving datagrams from the network. The blocking method `receive()` of `UDPDatagramConnection` will wait until a datagram is received and is stored into the given argument `datagram`, which is a `Datagram`. The received portion of bytes stored in `datagram` will be extracted to a new array called `datagramInBytes`, and this latter will be delivered to the SLEF layer by its `receive()` method.

```
while (true) {
  // receives the incoming datagram
  macBroadcast.getDatagramConnection().receive(datagram); // a blocking method
  byte[] datagramInBytes = new byte[datagram.getLength()];
  // copy the right portion of bytes to the returned array
  System.arraycopy(datagram.getData(),0,datagramInBytes,0,datagram.getLength());
  slef.receive(datagramInBytes);  // send to SLEF
}
```

5.2 J2SE implementation

5.2.1 MACBroadcast

In the `MACBroadcast`'s constructor method, we open a datagram socket `DatagramSocket` with `portNumber` given as the port number. We also call that socket `datagramConn`. The broadcasting address required now is now longer a `String`, but an `InetAddress`.

```
datagramConn = new DatagramSocket(portNumber);
inetAddress = InetAddress.getByName(broadcastAddress);
```

When there's a packet to send into the network, `send()` method will create a `DatagramPacket` and use `datagramConn` to send that packet to the broadcast address defined by `inetAddress`.

```
packet = new DatagramPacket(netMessageInBytes, netMessageInBytes.length,
                                                    inetAddress, portNumber);
datagramConn.send(packet);
```

5.2.2 MACReceiver

At this receving side, we maintain a `DatagramPacket` called `packet` to store the incoming datagram. Its creation requires a bytes array and the array's length. We fix this length to 1500, which is the standard size for a datagram. The thread also perfoms the blocking method `receive()` of the `DatagramSocket` class, and then it stores this incoming datagram into `packet`.

```
message = new byte[1500];
packet = new DatagramPacket(message, message.length);
macBroadcast.getDatagramConnection().receive(packet); // a blocking method
```

After that, we extract the received portion of bytes and store it into another array, which is delivered up to the SLEF layer.

```
byte[] bytes = new byte[packet.getLength()];
System.arraycopy(packet.getData(), 0, bytes, 0, bytes.length);
slef.receive(bytes);
```

Chapter 6

Deployment

In order to deploy the J2ME application on mobile phones, two files are required:

1. Java Archive file (JAR)
2. Java Application Descriptor file (JAD). This file contains information describing the application, such as the application's name, the JAR file size. This information belongs to manifest information within the JAR file. However, the fact that they live outside the JAR enables the AMS to learn about the application without installing it.

We provide two files PushToChat.jar and PushToChat.jad to be copied into the phone's directory. For the J2SE version, its JAR file can be directly executed.

The application has been tested with different device's combinations. Below we would like to give some observations and remarks:

1. When the J2ME version runs on laptops over the Java simulator, the connection is very low. It does not miss messages but takes a lot of time to receive and display them. The problem is solved when the J2SE version is used.
2. The time for a laptop's card to connect to the ad hoc network is superior to the time needed between Smartphones.
3. Currently, when the application starts, the sequence number will begin at 1 and is increased by 1 each time a new payload is sent. However, below is a scenario in which this causes a problem.

   Pierre is chatting and is sending his 94th message. Suddenly, his connection fails and he has to restart his application, then the sequence number will begin at 1. As a message is identified by its originator and its sequence number, Pierre's new 94 messages will not be delivered by other SLEF layer to their application layer, because they were received at least once. Hence, these 94 messages will not be displayed.

To avoid this situation, each time the application is launched, the starting sequence number must be bigger than the one at the previous exit. A solution is to take the current time value, as it is a long (64bits) and it continuously increases. However, because of the length of the number, it is preferable to not display it with the content of the message.

4. In order to demonstrate the functionalities of SLEF, we added a field in Payload called `receivedFrom` to indicate the node that transmitted this payload. If this node is different from the payload's originator, then we concatenate to the message displayed on the device screen the notice "*forwarded by*". This is because before delivering the payload up to the application, SLEF changes the value of this field to the value of the `NetMessage`'s `sender`. This action would violate the rule of the encapsulation, which would not change the content of the encapsulated object.

   Combining with the sequence number remark above, we provide another version in which the sequence number starts with the current time (therefore it is not displayed) and we remove the `receivedFrom` field from Payload class. Hence, this new version is a real chat application for end-users, where SLEF mechanism is hidden. The current version is still used for development purpose.

Chapter 7

Conclusion and Future work

The first part of the project concentrated on the learning of the new platform J2ME. It was not a straight line to understand how J2ME applications work and to find necessary information or functionalities that could be applied to the HTC S620. Several test applications were developed apart in order to find out the supported and unsupported functionalities of the phone. An important part of the application layer was already built during this period. The second part was dedicated to the understanding of the SLEF protocol. I received a lot of help from my assistant in this part. After having achieved necessary knowledge on J2ME and on SLEF, the program's architecture was designed followed by its development until this day.

As this point of the project, two objectives defined at the beginning of the project have been reached:

- SLEF is implemented on Smartphone. Almost all functions are supported.
- A chat application built over SLEF for Smartphone is released. This application also supports usual chat functionnalites (block/unblock users, a scrollable text area) and can diplay SLEF status.
- Moreover, a version in J2SE is also provided and is fully compatible with the J2ME version.

However, there still are missing components and here are some suggestions for the future work:

- Resolve the user's identifier problem discussed in section 3.2.6.
- The value of `pendingRetransmissionDelay`, used to create timers for packets sent in pending mode, and of `busyInterval`, used to compute the presence indicator, should be carrefully computed.
- Reduce the number of threads due to the use of timer for the packets.

In the final word, I would like to thank Alaeddine El Fawal for his kind guide and his useful advices during the whole semester. The project has been a pleasure. It certainly required a lot of work, but considering the experiences and knowledge acquired, it was worth for.

List of Figures

2.1 The Java Platform Architecture and the J2ME Layers 9
2.2 The J2ME MIDlet's life cycle 12
2.3 The J2ME's lcdui package 13
2.4 Architecture 15

3.1 The J2ME application's life cycle 17
3.2 Payload 18
3.3 J2ME Reception 19
3.4 J2ME Sending 19
3.5 Push To Chat, J2ME's user interface 21
3.6 Three buffers of the message reordering process 22
3.7 The messages reordering process upon reception 25
3.8 Push To Chat, J2ME's DisplayItem 26

4.1 NetMessage 34
4.2 The Epidemic buffer 37
4.3 SLEF Reception 38
4.4 Received NetMessage state 39

5.1 MAC layer's design 49

Bibliography

[1] Sing Li and Jonathan Knudsen, *Beginning J2ME, From Novice to Professional*

[2] Sun Developer Network, *J2ME Low-Level Network Programming with MIDP 2.0 by Qusay H. Mahmoud*, http://developers.sun.com/mobility/midp/articles/midp2network/

[3] Michael Juntao Yuan, *Enterprise J2ME, Developing mobile Java applications*

[4] Ibrahim El Ghandour, *Self Limiting Epidemic Forwarding Implementation*, Semester Project

[5] Alaeddine El Fawal, *Multi-hop Broadcast from Theory to Reality: Practical Design for Ad Hoc Networks*

[6] Vartan Piroumian, *Wireless J2ME Platform Programming*