Technical Documentation

Mobii

Geoffrey Aubert - geoffrey1.aubert@epitech.eu
Nicolas De-Thore - nicolas.de-thore@epitech.eu
Camille Gardet - camille.gardet@epitech.eu
Sebastien Guillerm - sebastien.guillerm@epitech.eu
Nicolas Laille - nicolas.laille@epitech.eu
Pierre Le - pierre.le@epitech.eu
Clement Morissard - clement.morissard@epitech.eu

Today, the smartphone market is booming, the number of model continues to grow and EVERY companies use its own platform for his product.

The user possess multiple telephone, or who have purchased a newer model but a different brand of its old model must juggle with the various softwares offered by manufacturers to manage the data in its phones.

Mobii offers these users a **single tool**. It may easily synchronize data, make backups, transfer its library from one phone to another one.

But not only! The user can also do without the phone when it is on his computer. The application *Mobii* on computer will receive and make calls, send and receive SMS via the phone.

Similar applications exist, but they are not cross platform or functionality limited, outdated or not updated since a while.

This document describes in detail the architecture of the components of the *Mobii* project. Here are the answers to questions like **How theses elements works ?**, **What are the interactions between these elements ?**

You will discovered:

- all the characteristics of different tiles of the project (technical, behavioral, functional, etc.),
- different aspects of the project (logic, process, data, quality, etc.),
- a vision for the implementation of these tiles.

Document's description

Title	EIP 2014 - Technical Documentation
Date	Thursday, 16st of January (01/16/2014)
Author	Mobii Group
Manager	Nicolas Laille
E-mail	mobiilabeip.epitech.eu
Subject	Technical documention for Mobii project
Version	Final

Version table

Date	Author	Version	Comments
03/21/2013	Group	1.0	Document creation
05/29/2013	Group	3.0	Document modification and merge
07/15/2013	Group	4.0	Document modification and merge
10/20/2013	Group	5.0	Document modification and merge
12/21/2013	Group	6.0	Document modification and merge
01/16/2014	Group	Final	Document modification and merge

Contents

Chapter 1

Mobii Server

1.1 Technologies and development environment

The server has been developed in C++ and the following libraries have been used:

- Boost 1.52.0 (Asio, Threads, bimap),
- Soci 3.2.0 (database layer).

1.2 Global Architecture

The server is mainly used to initiate and handle communications between the clients, and also to interact with the database.

The software use a third party library to process data from the database, this library is called SOCI, and supports many types of data in addition of being multiplatform. SOCI can also be used with Boost.

A third party network library called Boost asio allow a high level network abstraction, the benefits are the multiplatform management, its flexibility and a great documentation.

The server will be handled exclusively from an external client. So, it will be developed a monitoring infrastructure, separate from the server itself. This client will not be developed in this document.

In order to process incoming requests from the network layer, the server contains a “tail” of messages to execute, which will be handled by different threads.

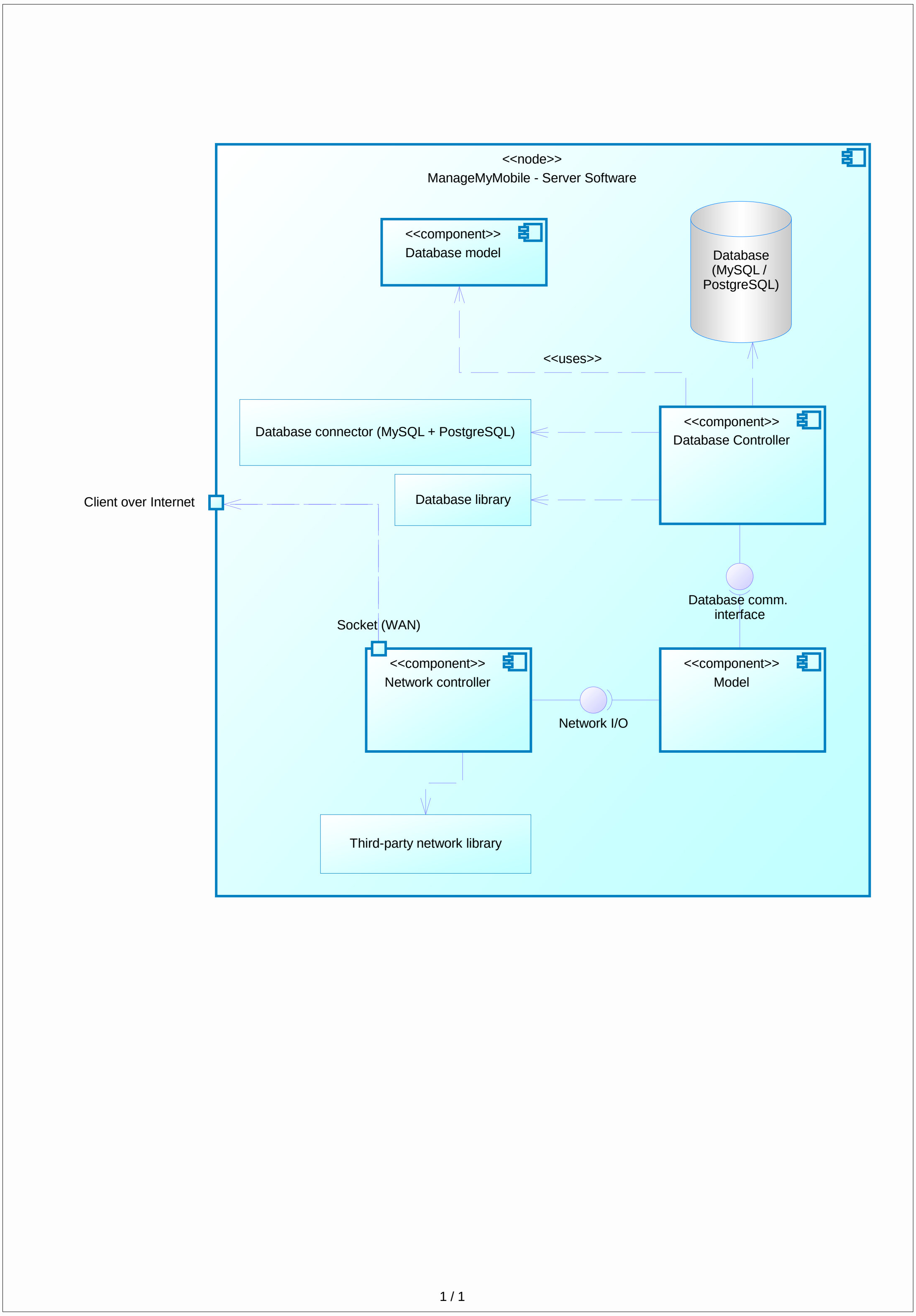


Figure 1.1: Mobii server's components interconnections

1.3 Architecture details

In this part, the architecture server-side will be more specified in order to correctly understand the relations between the software's classes and their respective roles.

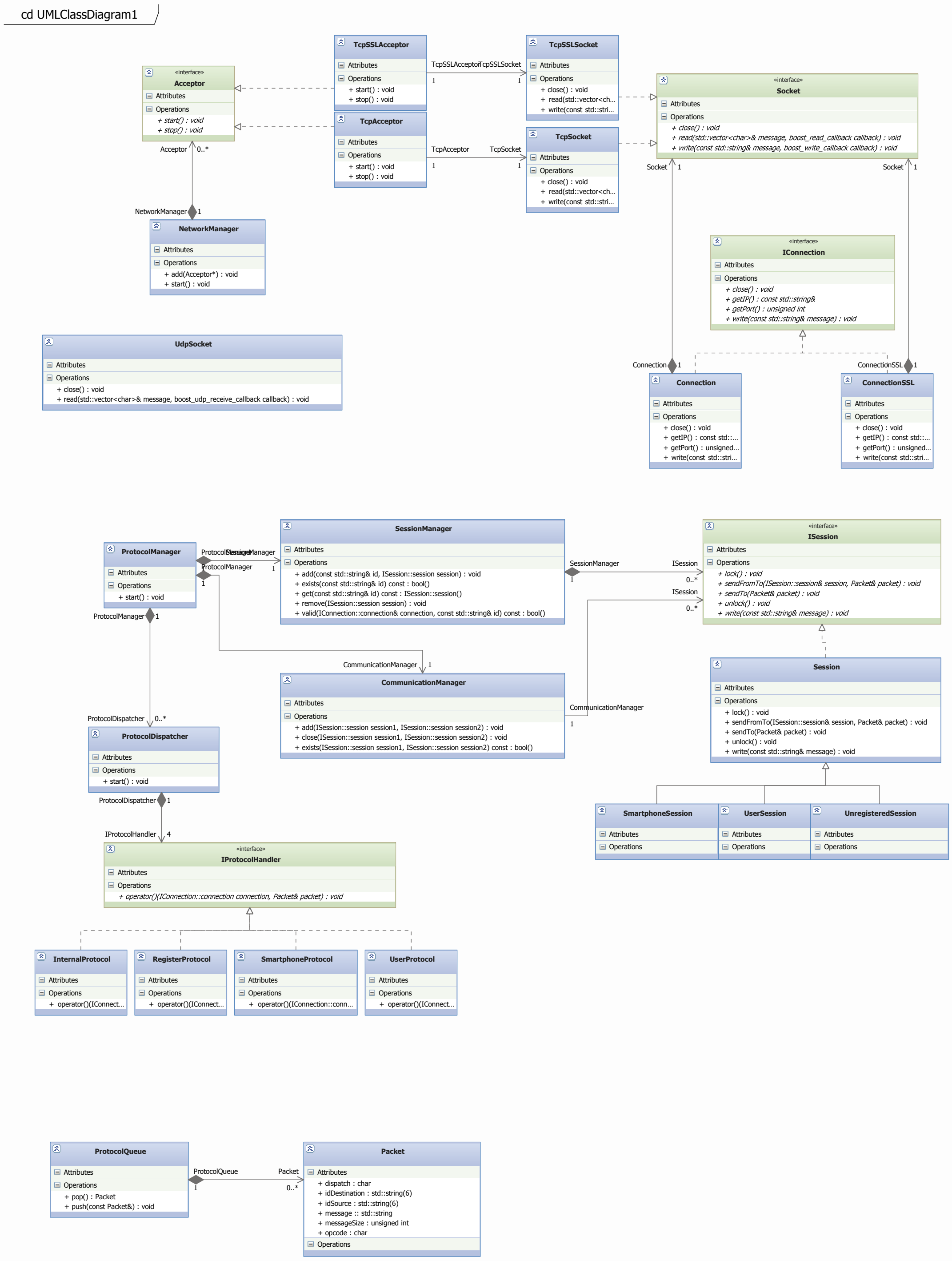


Figure 1.2: Mobii server's class diagram

1.3.1 Network layer architecture

A distinction has been made between TCP (SSL or not) and UDP sockets.

The acceptors allow us to initiate the TCP connections and create ones with the associates sockets (TcpSocket/TcpSSLSocket).

A socket is an abstraction of system sockets with the TCP protocol, it permits to read and write data using TCP.

A “Connection” is the representation of a connection on the server; it’s defined by a Socket, an IP address, and a Port. “Connection” handles the incoming and out coming messages. It places them in a queue and converts the incomings in Packet before packing them in the ProtocolQueue class.

UdpSocket is the abstraction of the system sockets with the UDP protocol; it allows reading and writing data using UDP.

1.3.2 Procotol management architecture

By the network layer received messages are unpacked in the ProtocolDispatcher class and then send to the correspondent IProtocolHandler.

Each IProtocolHandler (e.g. InternalProtocol, RegisterProtocol) is handling a part of protocol.

In the protocol part of the server, each Connection is represented by a Session which permits the link between a Connection and the ID of the Connection.

A Connection is considered like an UnregisteredSession as long as the Connection is not related to an ID. While a Connection is an UnregisteredSession, it can’t be used by the rest of the protocol.

SessionManager processes the Session, it allows to get a Session using his ID or his Connection, and it also allows to validate a Connection using an ID. A bimap is used to stock the Sessions.

CommunicationManager is a class which is able to handle communications between Sessions, it creates links between Sessions and verifies if two Sessions have the right to communication with each other.

Chapter 2

Mobii Desktop Client

This chapter is intended to provide details regarding the Mobii Platform's desktop client conception.

The general conception of the application will be presented first, then the detailed conception of each of its main modules, as well as their mutual interconnections.

Note — Detailed documentation about classes can be found at the project's official website, at the following address: http://eip.epitech.eu/2014/managemymobile/dev_doc/desktop/

2.1 Technologies and Development Environment

The Mobii desktop client is based on the Qt 5 framework. This choice was believed to be the best choice regarding the multiplatform aspect of the project, as one of our main goals is to offer this solution as many users as possible. It is also recommended to use the latest version of that framework, the 5.0.1 version.

Please note that the Mobii client application was previously developed using the Qt 4 framework. It has been decided to migrate to the newer Qt 5.x framework as it offered new functionalities which were considered interesting enough to justify the migration of the project.

The main development environment for this application is Microsoft Visual Studio 2010. That said, a QMake compilation solution is also provided for developers wishing to develop on another platform than Windows.

The graphical interface conception is based on Qt Designer. This tool is provided in the Qt 4 SDK.

2.2 Mobii Client Architecture Overview

The Mobii client application is based on the MVC design pattern, also known as Model-View-Controller. Using this pattern is made easier as this is the way Qt user interfaces are meant to work.

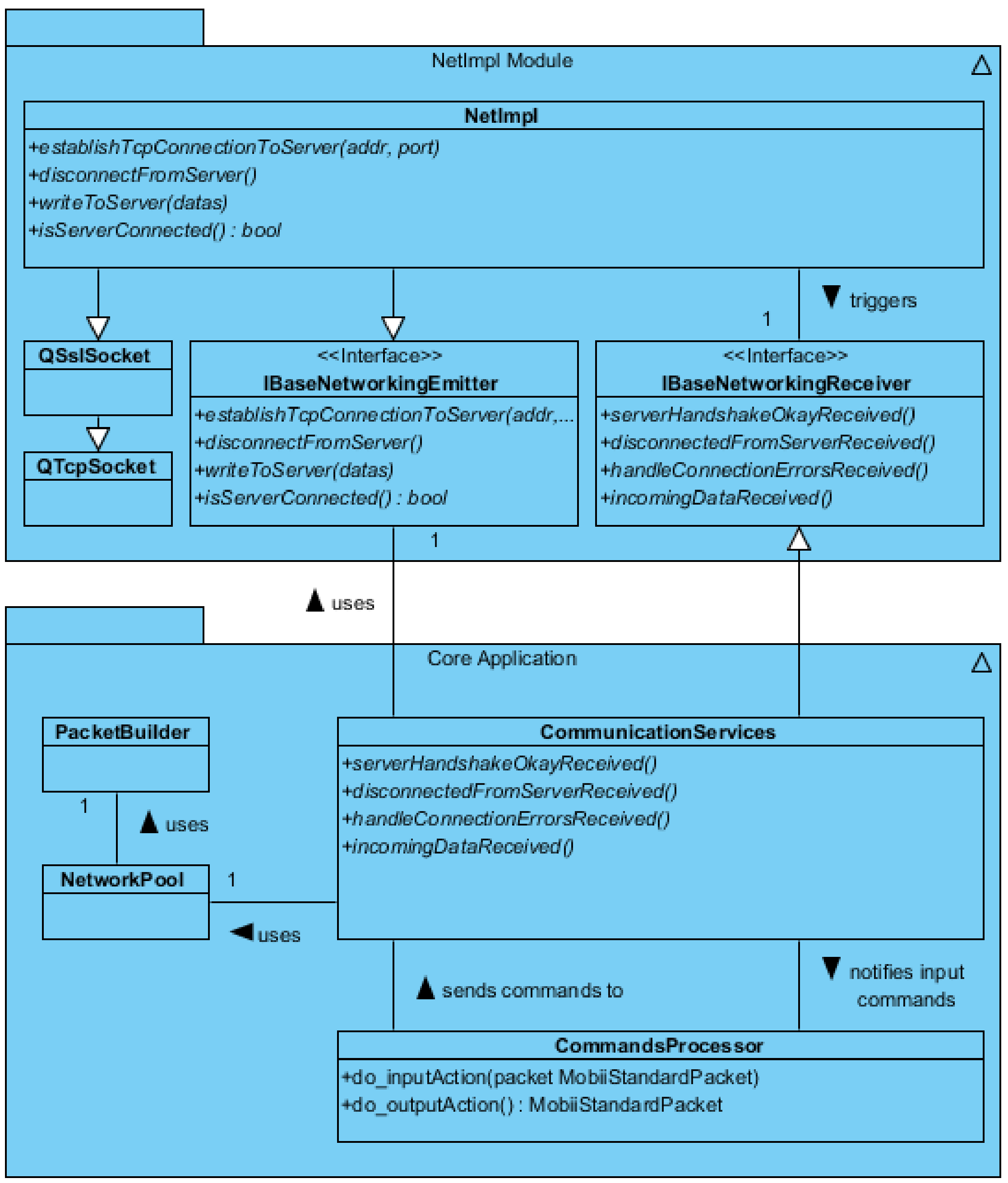

NetImpl Module
NetImpl
+establishTcpConnectionToServer(addr, port)
+disconnectFromServer()
+writeToServer(datas)
+isServerConnected() : bool
▼ triggers
1
QSslSocket
QTcpSocket
<<Interface>>
IBaseNetworkingEmitter
+establishTcpConnectionToServer(addr,...
+disconnectFromServer()
+writeToServer(datas)
+isServerConnected() : bool
<<Interface>>
IBaseNetworkingReceiver
+serverHandshakeOkayReceived()
+disconnectedFromServerReceived()
+handleConnectionErrorsReceived()
+incomingDataReceived()
1
▲ uses
Core Application
PacketBuilder
1
▲ uses
NetworkPool
1
◀uses
CommunicationServices
+serverHandshakeOkayReceived()
+disconnectedFromServerReceived()
+handleConnectionErrorsReceived()
+incomingDataReceived()
▲ sends commands to
▼ notifies input commands
CommandsProcessor
+do_inputAction(packet MobiiStandardPacket)
+do_outputAction() : MobiiStandardPacket

Figure 2.1: Mobii application's components interconnections

2.2.1 Notes about using of Qt's Network libraries

The module in charge of displaying datas on the screen as well as the one that handles network inputs and outputs use the appropriate Qt modules.

Even if the network module is based on Qt's network classes, it has been built so that one can decide to use another technology or implementation if necessary. Rather than interacting directly with the Qt-based network module, the Business Logic module uses the **IBaseNetworkingEmitter** and **IBaseNetworkingReceiver** interfaces.

2.2.2 Main modules

The **Display Module** (or *Views*) is the bridge between the user and the other modules of the application. It allows the user to input datas to be processed, as well as graphical representation of various internal events.

The **Business Logic** (or *Model*) is in charge of ensuring that the Mobii protocol is well applied. It handles input and output Mobii standard packets, whether they are received from a phone or the server.

The **Controller Module** handles interconnection between the **Display Module** and the **Business Logic** Module.

2.3 Detailed Architecture of the Mobii Application

This section details main notions and concepts, which rule the various application modules.

The following diagram sums up the interactions between the classes contained in the application:

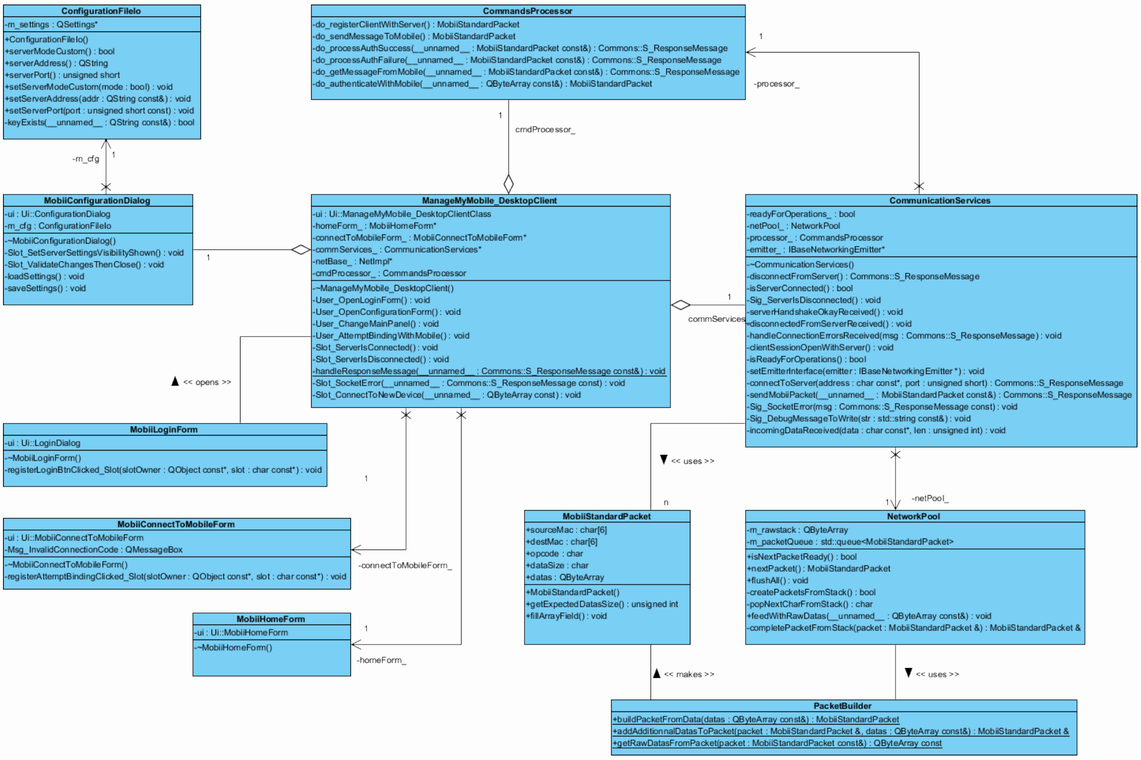

ConfigurationFileIo
-m_settings : QSettings*
+ConfigurationFileIo()
+serverModeCustom() : bool
+serverAddress() : QString
+serverPort() : unsigned short
+setServerModeCustom(mode : bool) : void
+setServerAddress(addr : QString const&) : void
+setServerPort(port : unsigned short const) : void
-keyExists(__unnamed__ : QString const&) : bool
CommandsProcessor
-do_registerClientWithServer() : MobiiStandardPacket
-do_sendMessageToMobile() : MobiiStandardPacket
-do_processAuthSuccess(__unnamed__ : MobiiStandardPacket const&) : Commons::S_ResponseMessage
-do_processAuthFailure(__unnamed__ : MobiiStandardPacket const&) : Commons::S_ResponseMessage
-do_getMessageFromMobile(__unnamed__ : MobiiStandardPacket const&) : Commons::S_ResponseMessage
-do_authenticateWithMobile(__unnamed__ : QByteArray const&) : MobiiStandardPacket
1
-processor_
1
cmdProcessor_
-m_cfg
1
MobiiConfigurationDialog
-ui : Ui::ConfigurationDialog
-m_cfg : ConfigurationFileIo
-~MobiiConfigurationDialog()
-Slot_SetServerSettingsVisibilityShown() : void
-Slot_ValidateChangesThenClose() : void
-loadSettings() : void
-saveSettings() : void
1
ManageMyMobile_DesktopClient
-ui : Ui::ManageMyMobile_DesktopClientClass
-homeForm_ : MobiiHomeForm*
-connectToMobileForm_ : MobiiConnectToMobileForm*
-commServices_ : CommunicationServices*
-netBase_ : NetImpl*
-cmdProcessor_ : CommandsProcessor
-~ManageMyMobile_DesktopClient()
-User_OpenLoginForm() : void
-User_OpenConfigurationForm() : void
-User_ChangeMainPanel() : void
-User_AttemptBindingWithMobile() : void
-Slot_ServerIsConnected() : void
-Slot_ServerIsDisconnected() : void
-handleResponseMessage(__unnamed__ : Commons::S_ResponseMessage const&) : void
-Slot_SocketError(__unnamed__ : Commons::S_ResponseMessage const) : void
-Slot_ConnectToNewDevice(__unnamed__ : QByteArray const) : void
1
commServices
CommunicationServices
-readyForOperations_ : bool
-netPool_ : NetworkPool
-processor_ : CommandsProcessor
-emitter_ : IBaseNetworkingEmitter*
-~CommunicationServices()
-disconnectFromServer() : Commons::S_ResponseMessage
-isServerConnected() : bool
-Sig_ServerIsDisconnected() : void
-serverHandshakeOkayReceived() : void
-disconnectedFromServerReceived() : void
-handleConnectionErrorsReceived(msg : Commons::S_ResponseMessage) : void
-clientSessionOpenWithServer() : void
-isReadyForOperations() : bool
-setEmitterInterface(emitter : IBaseNetworkingEmitter *) : void
-connectToServer(address : char const*, port : unsigned short) : Commons::S_ResponseMessage
-sendMobiiPacket(__unnamed__ : MobiiStandardPacket const&) : Commons::S_ResponseMessage
-Sig_SocketError(msg : Commons::S_ResponseMessage const) : void
-Sig_DebugMessageToWrite(str : std::string const&) : void
-incomingDataReceived(data : char const*, len : unsigned int) : void
▲ << opens >>
MobiiLoginForm
-ui : Ui::LoginDialog
-~MobiiLoginForm()
-registerLoginBtnClicked_Slot(slotOwner : QObject const*, slot : char const*) : void
1
▼ << uses >>
n
1
-netPool_
MobiiConnectToMobileForm
-ui : Ui::MobiiConnectToMobileForm
-Msg_InvalidConnectionCode : QMessageBox
-~MobiiConnectToMobileForm()
-registerAttemptBindingClicked_Slot(slotOwner : QObject const*, slot : char const*) : void
-connectToMobileForm_
MobiiStandardPacket
+sourceMac : char[6]
+destMac : char[6]
+opcode : char
+dataSize : char
+datas : QByteArray
+MobiiStandardPacket()
+getExpectedDatasSize() : unsigned int
+fillArrayField() : void
NetworkPool
-m_rawstack : QByteArray
-m_packetQueue : std::queue<MobiiStandardPacket>
+isNextPacketReady() : bool
+nextPacket() : MobiiStandardPacket
+flushAll() : void
-createPacketsFromStack() : bool
-popNextCharFromStack() : char
+feedWithRawDatas(__unnamed__ : QByteArray const&) : void
-completePacketFromStack(packet : MobiiStandardPacket &) : MobiiStandardPacket &
MobiiHomeForm
-ui : Ui::MobiiHomeForm
-~MobiiHomeForm()
1
-homeForm_
▲ << makes >>
▼ << uses >>
PacketBuilder
+buildPacketFromData(datas : QByteArray const&) : MobiiStandardPacket
+addAdditionnalDatasToPacket(packet : MobiiStandardPacket &, datas : QByteArray const&) : MobiiStandardPacket &
+getRawDatasFromPacket(packet : MobiiStandardPacket const&) : QByteArray const

Figure 2.2: Application-wide class diagram

2.3.1 Global events bus system

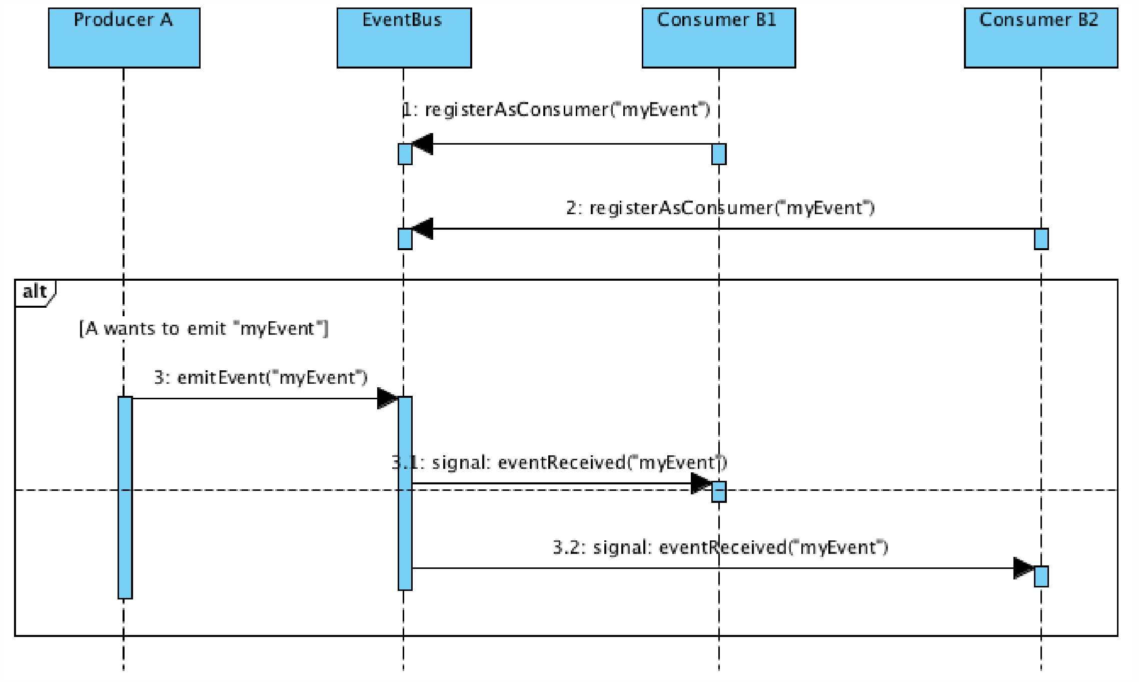

Producer A
EventBus
Consumer B1
Consumer B2
1: registerAsConsumer("myEvent")
2: registerAsConsumer("myEvent")
alt
[A wants to emit "myEvent"]
3: emitEvent("myEvent")
3.1: signal: eventReceived("myEvent")
3.2: signal: eventReceived("myEvent")

Figure 2.3: Typical use of the events bus system

Most of the application is based on semi-independent modules, thus making the client application's architecture decentralized.

In order to allow a proper intercommunication between components within the application, while preserving code isolation, an events bus system has been created.

This events bus is operated by the **EventBus** class. It uses the *Publisher/Consumer* design pattern. This allows module A to broadcast a specific message with arbitrary datas, while another module B can subscribe to the event triggered by A, allowing B to be informed of the broadcast of a new message.

Events transmission is based on Qt's signals and slots system. When an event is produced, a Qt signal is triggered. When an event is consumed, a Qt slot is activated.

2.3.2 Panels management within the User Interface

The philosophy of the Mobii user interface is to provide most of the application's features within an all-in-one integrated window. Such integration is ensured by a panels system that is able to selectively load a given panel within the main window, which associated class is **ManageMyMobile_DesktopClient**.

Each of these panels is made of one Qt Designer .UI view and one dedicated controller, which inherits QWidget. This local controller is in charge of handling user datas inputs. It also handles datas exchange with the Business Logic module, if applicable.

Moreover, all panels inherit the **IEventBusClient** interface, which allows them to gather or emit arbitrary events. Senders and receivers can be panels or any other module which

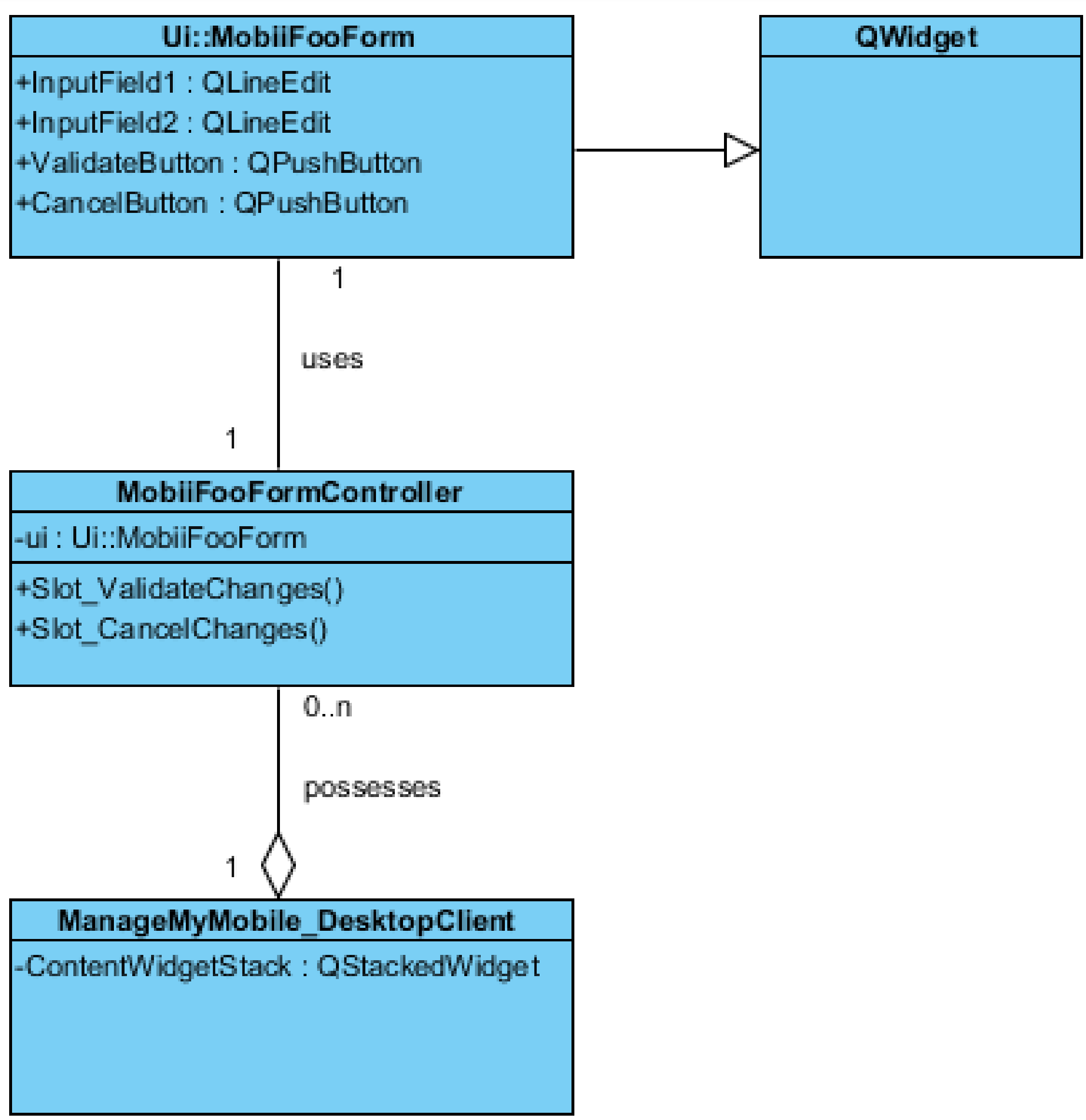


Figure 2.4: Composition of a typical panel

inherit the **IEventBusClient** interface.

The main controller class, **ManageMyMobile_DesktopClient**, groups all of these panels within a single object, **ContentStackWidget**. This object allows dynamic loading of the panel through a single method call.

The example shown above describes what classes a typical panel contains, as well as their mutual interconnections. The UI class only communicates with its local controller, which has a total and exclusive access to all member datas in the UI. It also handles and processes the events related to the signals sent by the view.

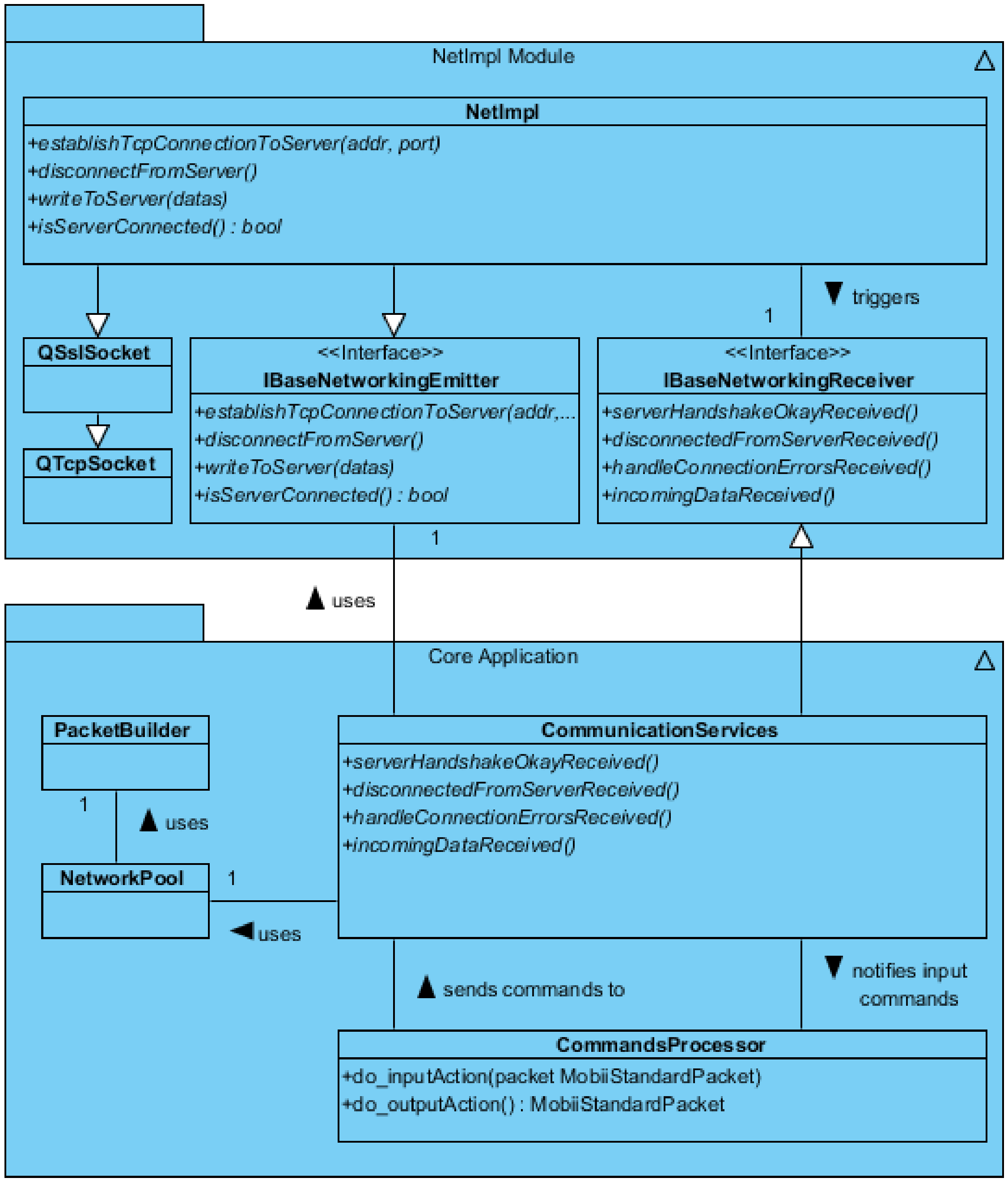

NetImpl Module
NetImpl
+establishTcpConnectionToServer(addr, port)
+disconnectFromServer()
+writeToServer(datas)
+isServerConnected() : bool
triggers
1
QSslSocket
QTcpSocket
<<Interface>>
IBaseNetworkingEmitter
+establishTcpConnectionToServer(addr,...
+disconnectFromServer()
+writeToServer(datas)
+isServerConnected() : bool
<<Interface>>
IBaseNetworkingReceiver
+serverHandshakeOkayReceived()
+disconnectedFromServerReceived()
+handleConnectionErrorsReceived()
+incomingDataReceived()
1
uses
Core Application
PacketBuilder
1
uses
NetworkPool
1
uses
CommunicationServices
+serverHandshakeOkayReceived()
+disconnectedFromServerReceived()
+handleConnectionErrorsReceived()
+incomingDataReceived()
sends commands to
notifies input commands
CommandsProcessor
+do_inputAction(packet MobiiStandardPacket)
+do_outputAction() : MobiiStandardPacket

Figure 2.5: Network communications architecture

2.3.3 Network Architecture and Business Logic

2.3.3.1 Low-level Protocol

The low-level protocol management is the responsibility of the **CommunicationServices** class, which is the central module in charge of handling all protocol-related events.

It is a main component of the Business Logique module and serves as a bridge between all network-related operations and protocol management.

This protocol is one of the two being used within the Mobii ecosystem. It is qualified as a "low-level" protocol because it is the closest to the system network layer.

It is mainly a transporter protocol which implements mechanisms allowing all elements of the platform to transfer useful datas between each other.

These mechanisms are the following:

- **Packet sender (unique identifier)**
- **Packet receiver (unique identifier)**
- **Order to execute on the receiver side**
- **Expected size of the useful load**
- **Packet's useful load**

2.3.3.2 Low-level input/outputs

Low-level network input and outputs are handled by an external module, **NetImpl**, which is not part of the main project (*Core Application* in the above diagram).

These two modules communicate through two interfaces, **IBaseNetworkingEmitter** and **IBaseNetworkingReceiver**.

Packets sent from the network are reconstructed using the aggregation class **NetworkPool**. Any Mobii standard packet is built using **PacketBuilder**.

2.3.3.3 Actions Protocol

The Actions Protocol is the second protocol of the Mobii Platform. It is designed to be the useful load within a "low-level" Mobii packet.

This protocol is specialized in business informations transferring, especially between a desktop client and a Mobii mobile app.

The JSON standard - or JavaScript Object Notation - has been selected to implement the Actions Protocol. It is a flexible language, thus allowing the Actions Protocol format to be reshaped depending on business needs. This avoids modifying the underlying "low-level" protocol.

Another reason why JSON was chosen is that it is a wide-spread standard, for which a lot of reliable implementations exist for most of the existing platforms.

Even though the Actions Protocol is naturally flexible, any JSON structure which has been generated by the Actions Protocol must contain a *action* field, which defines the business meaning of the datas included in the packet.

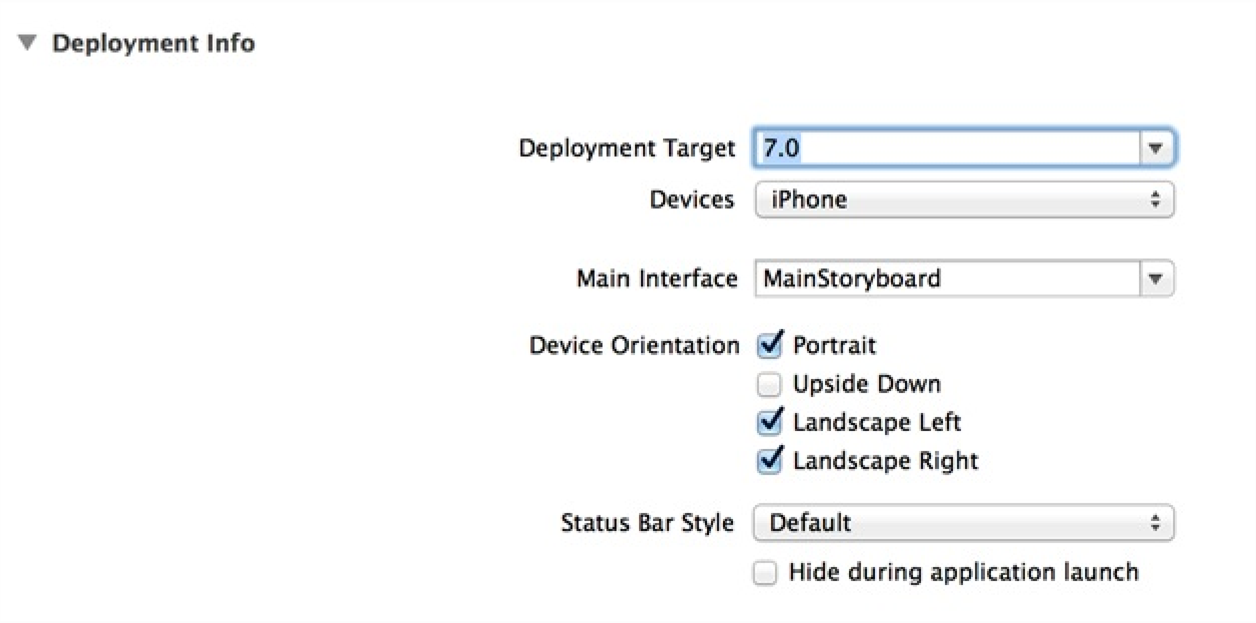
Chapter 3

Mobii iOS App

3.1 Project compilation

3.1.1 XCode

The Mobii iOS XCode project should be compiled using the latest version of XCode(v5.02) and the iOS 7.0 base SDK.



▼ Deployment Info

Deployment Target: 7.0
Devices: iPhone
Main Interface: MainStoryboard
Device Orientation: ☑ Portrait
□ Upside Down
☑ Landscape Left
☑ Landscape Right
Status Bar Style: Default
□ Hide during application launch

3.1.2 Dependencies

The project includes external libraries :

- **Cocoa Async Socket for Mac and iOS**. This library provides tools to manipulate sockets in an asynchronous way. Two objects have been included to the project

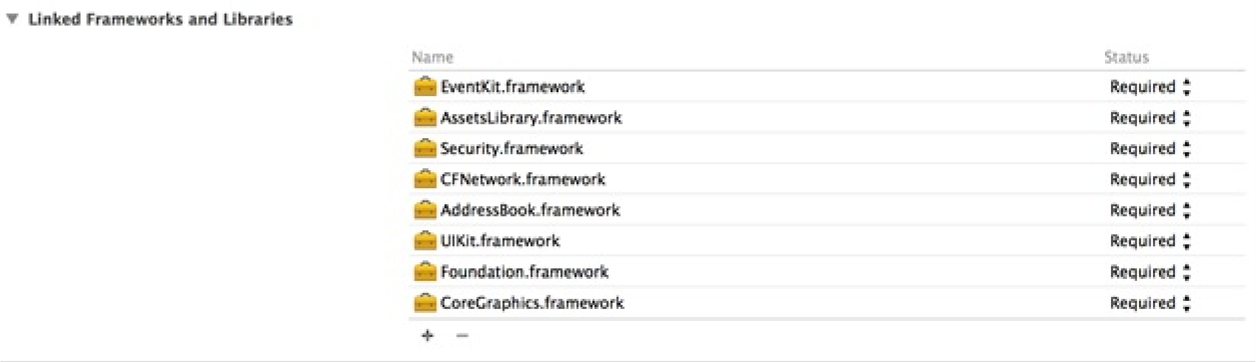
GCDAsyncSocket and *GCDAsyncUdpSocket*. Find more informations on : http://github.com/robbiehanson/CocoaAsyncSocket

- **Touch JSON**. This library provides a handy set of tools to manipulate JSON data. Two object have been included from this library: *CJSONSerializer* and *CJSONDeserializer*. Find more informations on : https://github.com/TouchCode/TouchJSON

It also relies on non-default frameworks :

- EventKit
- AssetsLibrary
- Security
- CFNetwork
- AddressBook

All those frameworks are included within thx XCode SDK but should be linked during the compilation process. To do so, add those to the Linked Frameworks and Libraries in the project property.



▼ Linked Frameworks and Libraries

Name	Status
EventKit.framework	Required
AssetsLibrary.framework	Required
Security.framework	Required
CFNetwork.framework	Required
AddressBook.framework	Required
UIKit.framework	Required
Foundation.framework	Required
CoreGraphics.framework	Required

3.2 App structure

The Mobii iOS App follows a MVC Design Pattern. Most of the mechanic of the app occurs in 5 main model object.

3.2.1 Networking

All the networking aspect of the project is handled in the *ConnectionManager* object. It is created with some connection details (IP, pot, etc.) and encapsulate the type of socked used. A set of useful methods is provided to interact with the connection in the most abstract manner.

3.2.2 Contacts management

The *ContactManager* object encapsulates the *AddressBook* Framework and provide a set of methods to retrieve and manipulate the contact list. The expected format of a contact is specified by the action protocol.

3.2.3 Calendar management

The *CalendarManager* object Encapsulate the EKEvent Framework and provide a set of methods to retrieve and manipulate the calendar. The expected format of an event is specified by the action protocol.

3.2.4 Files management

The FileManager object Encapsulate the ALAsset Framework and provide a set of methods to retrieve and manipulate the files (for now, just pictures).

3.2.5 Packet Management

The PacketManager object handles the packet received by the client app and generates answer packets. This, it contains each one of the previous objects to get the appropriate data depending on the request. The packet format is defined in the communication protocol.

3.3 Issues

3.3.1 Known Bugs

3.3.2 To-Do

- Internationalization
- Local notification when contact/evend added
- Plist file to match client label and iOS label for phone or email entry

Chapter 4

Mobii Android App

4.1 Technologies and development environment

The Android app is developed in Java with Eclipse (Android Bundle).
App tests are performed with an emulator and on a smartphone Samsung Galaxy S3.

4.2 Overall architecture

The app is mainly composed of two activities and one service.

4.3 Detailed architecture

4.3.1 Activities

The main activity allows the connection between the app and the server just by a click on the connection button. It is almost composed of an menu with which we can access to a settings pannel ("Settings") and to the Mobii website link ("About") and the stop of the app ("Exit").

Once the connection is established, a second activity is launched. This one is composed of a button to close the connection between the app and the server.

4.3.2 Connection service

A service (ConnectionService) is launched when connecting to the server to create a socket and maintain the connection between the application and the server, even in background. With the socket it is possible to send and receive data streams.

4.3.3 Management of inbound and outbound streams

A module composed of a main class (TcpPacket) manages the creation of packets to send, and the reception and the handling of the inbound packets.

4.4 Features description

The Android app allows to:

- Get smartphone information, as the brand, the name, the OS version,
- Manage contacts (recovery, add, delete and update),
- Send and receive sms from the client.

The following features are in progress:

- Manage calendar (recovery, add, delete and update),
- Send and receive mms,
- Get multimedia content, as musics, videos, pictures and other files.

The add of some features could be also interesting:

- Call and receive call from the client, if the connection allows it (3g/4g and Wi-Fi).

4.5 Libraries

No externally library is used.

However some have been tested:

- Google gson, for the format of the messages composing the packets,
- Joda-time, for the management of dates/formats of the dates.

A library could also be tested to try to improve connection part:

- Extasys.

Chapter 5

Mobii Windows Phone App

5.1 Technologies and Development Environment

The application has been developed with Visual Studio 2013, running under Windows 8.1 with the Windows Phone Development Kit 8.1. The project may present compatibility issue with ulterior version of these tools.

The language used are C++ for the connection part, C# and XAML for the rest.

5.2 Externals dependencies

JSON.NET is the only external libraries used, and is included in the project.

More about JSON.NET: https://json.codeplex.com/

5.3 Architecture

The Mobii WP application's design use the MVVM model, and a specially created internal library for the networking.

5.3.1 Networking

This module is written in C++ and use WinSock library, to keep a great freedom on the socket's manipulation.

5.3.2 Contacts and Calendar

Since the SDK does not allow to modify, create or delete *directly* the contact or the calendar, we create a ContactStore, which will be a local copy of the contact and calendar.

The information (read-only) are imported directly from the Microsoft.Phone.UserData API.