**UNIVERSITY OF OSLO**
**Department of Informatics**

# PISA - The Platform Independent Sensor Application

Michael Andre Krog

**August 1, 2014**

# Abstract

This thesis focuses on making a platform independent application for reporting observations and positions of users/units moving out in the terrain. This application is intended for use on smart devices, i.e., smartphones and tablets. It is especially the iOS and Android operating systems (OS's) we focus on, which are respectively owned and managed by Apple and Google. What we want is for the application to work in Disconnected, Intermittent, Limited (DIL) environments. Simply speaking, DIL environments are networks where the user will experience loosing the connection from time to time, and find the network to be quite slow or unresponsive.

We focus on running our application on iOS and Android devices in a military defense context, due to the fact that these civilian commercial off-the-shelf (COTS) mobile devices are cheap, yet very powerful sensor platforms. These are also available at a much lower cost than military hardware. Additionally, a lot of users are already familiar with such devices, and need little training in how to operate them. This makes a platform independent approach quite reasonable.

Several approaches for implementing this kind of application already exist. However, in this thesis we use the PhoneGap framework to add the concept of platform independence. Most of the earlier research on operating in DIL environments focus on using SOAP-based Web services for communication purposes. In this thesis, we use the Representational State Transfer (REST) communication pattern, which is not extensively done before. We also connect our application to an existing proprietary solution, Collective Environment Interpretation (CEI). In addition, we demonstrate how our solution can be used towards a SOAP-based infrastructure with a wrapper server.

The evaluation of our experiments yielded both promising and successful results. The majority of testing was carried out on the functionality and performance of our application. In the DIL environments, we experienced either complete success or limited success. All in all, the tests were successful. The Graphical User Interface (GUI) was also tested, and it resulted in quite constructive feedback along with suggestions of potential improvements.

# Preface

This thesis was written as a part of my Master's degree in Computer Science at the University of Oslo (UiO), Faculty of Mathematics and Natural Sciences, Department of Informatics. The thesis was written at the Norwegian Defence Research Establishment (FFI) and the University Graduate Center (UNIK).

I would like to express my sincerest gratitude by thanking my academic supervisors at FFI, Dr. Frank Trethan Johnsen and Cand.Scient. Trude Hafsøe Bloebaum, for their impeccable counseling, support and numerous advice throughout the work on this thesis. The assistance provided by my supervisor at UiO, Prof. Dr. Carsten Griwodz, is also much appreciated. Thanks also go to Dr. Bård K. Reitan at FFI, for great assistance in help with the CEI integration. Additionally, I would like to extend my thanks to Kristoffer Karud, for providing and helping with the testing tool used in this thesis. The assistance received from testing the user interface of our application is greatly appreciated, so thanks again go to Kristoffer Karud, along with Anniken S. Hofgaard and Dr. Ketil Lund.

I would also like to thank my wonderful friends and dear family for their huge support in these occasional stressful and hard times. A special thanks is extended to Team Rubberduck, for the much appreciated friendship during this work. Lastly, the finalization of this thesis would have been impossible without the most treasured support from Maren Amalie Halsen, and for always being there.

*Michael Andre Krog*

*Oslo, August 2014*

This page is intentionally left blank.

# Contents

This page is intentionally left blank.

# List of Figures

This page is intentionally left blank.

# List of Tables

This page is intentionally left blank.

# Listings

This page is intentionally left blank.

# Chapter 1

# Introduction

This thesis investigates the concept of using readily available devices, combined with standards, for use by military units, and others with similar needs (search and rescue, etc.). Therefore we have come up with a prototype application, i.e., the Platform Independent Sensor Application (PISA).

The focus of prior projects within the defense sector was primarily on proprietary and binary defense specifications. However, after the North Atlantic Treaty Organization (NATO) Network Enabled Capability (NNEC) Feasibility Study [1] in 2005, the focus changed to provide commercial off-the-shelf (COTS) solutions by using more open, interoperable standards, and often civil standards. The study found that Web services was the key enabling technology for achieving this goal.

This thesis looks at the use of COTS-devices in a way that we maintain compatibility with Web services, at the same time as we utilize the opportunities that these cheap devices offer.

## 1.1 Background and Motivation

Over the last few years there has been an increase in the focus on, and use of, commercial mobile devices within the defense sector. This development is largely due to the fact that current mobile devices are small, light and very powerful mobile sensor platforms that are available at a much lower cost than special purpose military hardware. Additionally, the end users are often already familiar with such devices, which enables them to be put to use with little additional training.

Most previous efforts related to the use of these mobile devices in a military context have focused on one specific operating system (OS) or device type (smartphone or tablet).

In addition, these efforts have been focused on developing one stand-alone system, and have only to a lesser degree focused on interoperability with other emerging standards and technologies being implemented within the defense sector in general, and NATO in particular. A few concrete examples of related work are presented in section 2.1. In the work presented in this thesis, we address several of these issues by looking into the development of an application that is platform independent (or cross-platform), functions on both smartphones and tablets, and seeks to achieve interoperability both with already existing systems and with the emerging standards being identified by NATO.

The NNEC program is the Alliance's ability to federate various capabilities at all levels, military (strategic to tactical) and civilian, through an information infrastructure. Networking and Information Infrastructure (NII) [9] is an infrastructure developed by NATO, and it is used for realizing NNEC. NII is the supporting structure that enables collaboration and information sharing among users, while reducing the decision-cycle time. This infrastructure also enables the connection of existing networks in an agile and seamless manner. The vision of NNEC is that people interacting with each other and sharing information will lead to better situational awareness and faster decision making. This ultimately saves lives, resources and improves collaboration between nations.

A well-known concept in the field of networking research is the Disconnected, Intermittent, Limited (DIL) environment [41,44]. A DIL environment is a common description of a network environment characterized by the constant possibility of periodic communication disruptions, bad connectivity, and limitation problems when it comes to network node capabilities (battery, storage, computational overhead, etc.).

When it comes to developing applications for handheld and mobile devices in DIL environments, we encounter several problems. The first major problem is the battery life of the devices. These types of reporting applications on smartphones and tablets have been tested out in the field before, and the experiences have proved the battery to deplete rather quickly when using network- and GPS features in dense intervals. Another major problem is the common disruptions in network state in DIL environments. Losing connectivity could mean loss of data, and this must be prevented. Communication must be both reliable and robust. If essential data is lost, the consequences could be drastic.

Current research concerning operations in DIL environments mainly focus on the use of SOAP-based Web services for communication purposes [6]. This is a major challenge due to the fact that SOAP is not natively supported on Android, one of the targeted platforms. In general there are few existing full SOAP implementations for devices like iOS or Android. Therefore, Representational State Transfer (REST) [8] is used as an alternative to SOAP in this thesis. REST is a much used architectural pattern, especially in civilian use, to transmit data over a connection between a client and a server. For the development of our prototype application (app), PISA, civilian technologies are used, and the already simple and familiar use of REST makes it only more convenient for us to use REST instead of SOAP. However, choosing REST over SOAP also provides a new challenge. Using REST in DIL environments has at the time of writing not been done extensively before, and does therefore not necessarily work out of the box. This must be researched further to find a suitable and satisfactory solution.

As previously mentioned, mechanisms must be applied to provide robustness against network disruptions and also reliable delivery of critical data. This is necessary to reduce the possibilities of losing vital data, thus some form of fail-safe mechanism needs to be implemented. With the use of SOAP in previous work, specific protocols may in some cases be applied to ensure reliable data delivery. In [6], protocols like Hypertext Transfer Protocol (HTTP) or HTTP Secure (HTTPS), User Datagram Protocol (UDP) and Advanced Message Queuing Protocol (AMQP), specifically the RabbitMQ implementation, are tested as transport methods with SOAP to provide reliable messaging. With REST, we do not have the same wide range of potential to accomplish these desired effects (see section 2.5). An efficient way of limiting data loss must be found. Using caching on the client side to back up essential data, when trying to transmit over a network with frequent disruptions, could be such a solution.

The mobile devices and platforms we implement for, are relatively inexpensive and easy to use, compared to military hardware. Devices running iOS and Android are easy to use in the way that they are highly common for everyday people, and therefore close to zero training is necessary for testing of the application on the devices. This also assumes that PISA is simple to use, which of course is intended. Even though we in this thesis develop the application with civilian technologies, and on mobile civil devices, usage is initially aimed at tactical units out in the field. This leads us to a concept called the *tactical edge.*

Dandashi et al. [4] define the tactical edge based on both a *user perspective* and a *technology perspective.* From a user perspective, the tactical edge are users that are warfighters directly involved in executing the mission. Here, “users” are those executing the mission in a forward deployed position. From a technology perspective, the tactical edge is where users operate in environments that are constrained by such things as limited communications connectivity and limited storage availability, e.g. like the aforementioned DIL environments. In addition, we also use civil technologies in the military field of application. In this thesis, we essentially look at using civil technology at the tactical edge.

## 1.2 Problem Statement

Reporting systems implemented on handheld devices are a particularly hot topic these days, and state-of-the-art solutions in this field are highly sought after and being researched. Previous work has only been focusing on developing solutions for a single OS, with Android being very popular (see, e.g., section 2.1). Compared to the platform scope, we take it a step further. We implement a platform independent reporting/sensor application for *both* iOS and Android, all written with the same exact code. This requires a cross-platform development framework, to make all features and functionality available across the different platforms.

All in all, the goal of this thesis is to research the possibilities of implementing a platform independent reporting system, that targets handheld devices in the tactical edge. The solution must also operate seamlessly in a DIL environment and handle frequent network disruptions in the most efficient way. The resulting findings are presented as a functional proof-of-concept prototype. And for all these reasons, we have given our application the fitting name "the Platform Independent Sensor Application", i.e., PISA.

## 1.3 Premises

The initial premises of this thesis is that the framework we use for implementation must be *platform independent*. The applied framework must support, and our application must work with, the major OS's. In our case we refer to Apple's *iOS* and Google's *Android*. These two are the main OS's to be developed for, and tested in this thesis. Both iOS and Android are widely known and much used platforms. In addition, the framework we find is required to be a *freely available, non-commercial framework*. This is to avoid costly experiments. The framework needs to provide desired results, so we can end up with a proof-of-concept prototype application, i.e., PISA.

## 1.4 Scope and Limitations

The goal of this thesis is to develop a platform independent application for reporting sensor data, such as observations and positions. More specifically, the user should be able to report and track his location, along with having the opportunity to take a photo of an incident nearby and send it to a central server which manages the observed data and positions. This implemented application should end up being a functioning prototype for use in national and coalition experiments.

We also want to facilitate interoperability with NII and SOAP Web services back-end systems, by utilizing standards when possible. This is desirable because further compatibility with NATO and Web service systems is made easier. This is important due to the fact that military solutions often use SOAP and Web services as the applied technology. For this prototype to easier work with other military systems in the future, the interoperability aspect is quite necessary to include on different levels (device platform, technologies, infrastructure).

The server we implement, on the other hand, is very basic. This server implements the absolute minimum functionality, to make it work properly with our application, PISA. This is due to the fact that the primary focus of the implementation work remains on the application side, *not* the server side. However, we also have made our prototype communicate with another server, one from a different FFI prototype system. This is to prove our application's potential of interoperability with other existing solutions.

Security is beyond the scope of this thesis. We use existing mechanisms and do not focus on including the security aspect ourselves on any levels in our prototype. We recognize that security is an important factor with many systems, especially military ones. However, due to time limitations we do not integrate any security solutions with PISA. This is left for future work.

## 1.5 Research Methodology

Denning [5] states that computer science is founded on four principles: Mathematics, science, engineering and computer disciplines (with the latter emerging in the early 1940s). He further specifies three main approaches for the computer discipline: Theory, abstraction and design. The last approach, design, is the one relevant for this thesis. It is based on the engineering discipline, and its process contains four steps. The steps are shown below in a bold font, and to the right of them are the corresponding methods applied in this thesis.

**Perform requirements analysis** - We have several premises for this thesis, and following the discussion in chapter 2, the final requirements for this thesis are specified in section 2.3.

**Derive a specification based on these requirements** We need to figure out what PISA should look like, and especially what functionality it needs to include. If previous and similar research exists, we want to build on these experiences and design the application to its fullest potential within the time frames. This stage is mainly defined in chapter 2, but also in chapter 3, section 3.2.

**Design and implement the system** - The requirements are set, a specification is defined, and in this step the application is designed and implemented. This is the process where the actual application is developed from the ground up. The corresponding chapter for this stage is chapter 3.

**Test the system** - To be able to state that the created prototype is a fully functional proof-of-concept with satisfactory behavior, it needs to be tested to see that it actually works the way it is supposed to. Testing and evaluation are described in chapter 4.

## 1.6 Contribution

The contribution of this thesis mainly consists of a platform independent prototype app that can be used in experiments by the Norwegian Defence. This has not been focused on before, and provides new knowledge of the existing possibilities of platform independent development. During the work of the thesis, we get an insight into the use of a civil platform in the tactical edge from a technology perspective. Civil technologies are used, along with civil devices like iOS smartphones and Android tablets.

Our solution not only includes a prototype application (PISA), but also a server component. This server can be used as a shell or wrapper towards SOAP Web services backend infrastructures, and provides interoperability with NII. By focusing on the server offering interoperability against NII, future work with this prototype, and communication between our wrapper server and Web services infrastructure is made easier.

The source code of the developed application is not available to the public, and therefore not accessible on for instance Git. This is due to distribution restrictions. The prototype does not work with any classified information, but it is not classified as releasable to Internet either since we operate with NATO standard formats and data. For these reasons the code is delivered on a flash drive handed to FFI. The distribution of the code is dictated by them, and interested parties from other NATO countries may optionally receive the source code.

## 1.7 Outline

The remainder of this thesis is structured as follows:

**Chapter 2** provides background information on the thesis, along with earlier work, the requirements analysis and important technologies and frameworks.

**Chapter 3** presents the design and implementation of the prototype solution.

**Chapter 4** evaluates the proposed solution, based on testing carried out in the thesis.

**Chapter 5** concludes this thesis, and discusses the aspects of design and implementation that are left for future work.

This page is intentionally left blank.

# Chapter 2

# Background and Requirements Analysis

In this chapter, we cover subjects like relevant earlier work, state of the art technologies, available frameworks for our objective and discussion of military operational requirements which lead to the applicable requirements for this prototype solution. Basically, we provide the background of this thesis along with the essential requirements analysis.

## 2.1 Related Work

In recent years there has been much military research on handheld tactical systems and how they work in disadvantaged tactical networks, i.e., DIL environments. Our goal is to implement a tactical reporting system consisting of a handheld application and a basic server. Below, we examine some important and relevant work for this thesis, explaining the similarities and differences.

The authors of [42] describe a set of prototypes that demonstrate the use of Web services in collaboration with SOAP in tactical environments. The users here are employing handheld devices to obtain situational awareness data, i.e., from observations. The testing was performed with Android devices exclusively.

Smartphone-Assisted Readiness, Command and Control System (SPARCCS) [3] is a system that utilizes smartphones in conjunction with cloud computing to extend the benefits of collaborative maps to mobile users while simultaneously ensuring that the command centers receive accurate and up-to-date reports from the field. Their main focus is on military and civil-military operations, and the devices used all run the Android OS. The report focuses on the difficulties on receiving reports from the field entities as well as ensuring these entities also have good situational awareness.

The Tactical Ground Reporting System (TIGR) [7] is a cloud-based application that facilitates collaboration and information sharing at the patrol level (tactical units in the field). The system enables collection and dissemination of fine-grained information on people, places, and events. TIGR offers a media-rich view of the battlefield with digital photos, videos, and high-resolution geo-spatial imagery. This is a fully tested system in real life scenarios, and it has proven to be very successful in multiple theaters of war with minimal impact on disadvantaged networks. This system employs military technology, and does not focus on civilian use.

There is also a report [40] that looks at the mobile complex in the military setting. The mobile complex consists of carried devices like smartphones and tablets, the networks such devices use, the (mobile) Internet with all its services, including the increasingly digital and technology competent users. This complex is, in the non-military world, displaying new ways of doing things. In the report they are conducting experiments searching for insight into how the military may relate to the mobile complex. That is, how to exploit the possibilities and reducing the risks. Their hypothesis is that the mobile complex will be an integral part of future command and control (C2) arrangements.

The tactical reporting system Collective Environment Interpretation (CEI) [10] is one of the most relevant reports for the thesis. The focus is directed at the tactical edge, and civilian technologies are employed. REST is used instead of SOAP, and the Internet is utilized as the network. Only Android devices were used as mobile units in the system. However, the similarities are definitely there. Because of this, the CEI system is later integrated with our own prototype solution.

Still, none of the previously mentioned research has focused on a platform independent solution. We consider interoperability as important, therefore we also include our focus on NII and a Web services proxy. This is where we come in. The next section extracts the essence of CEI and its work, explaining the relevance of the system and their experiences for this thesis.

## 2.2 A Discussion on CEI

CEI [10] is a “social tactical reporting system” which is intended to strengthen collective understanding and interpretation of situations. The prototype system consists of a mobile application, a server, a web application and a small scripting language. The system is developed to demonstrate smartphone technologies and related technologies with relevance for the Norwegian Armed Forces. The CEI-application is a map application intended for use with smartphones and tablets. The application lets users share their *positions* and *observations*, including the possibility for text and images, in a uniform way to all users of the CEI-service.

CEI was first used in an experiment in 2009, in order to demonstrate how mechanisms that are frequently used on the web, and especially in social technologies, could be applied to systems of military use. The CEI system employs civilian technologies and is designed for web and smartphones, and it uses internet/mobile internet as the network. The technology is therefore not as robust as a lot of other defense technology. Also, the mobile units run out of power pretty fast, and the majority of them have poor resistance against humidity, cold and impacts. Smartphones and tablets have the advantage that the technology is easy to use, and is often familiar to most people. This, along with the fact that many already own such a device, means that the technology can quickly be put to use, and requires little to no training. The technology is also relatively inexpensive compared to the large amount of functionality the devices have.

The CEI application can be installed on smartphones, tablets, computers and other units that run the Android OS. A web UI also exists which can be used through a browser for devices that don't support this OS, like ordinary computers. To gain access to and share information, one must provide a username and password for login on the application. Your credentials are verified by the server, and all users must therefore be registered there. Users have the possibility to share information such as *positions* and *observations*. Positions are shared to tell others about your current position, and one can choose to share a single position, or let yourself be tracked over time. Observations are positions with extra information and can describe incidents, places, buildings, or other information related to a geographic point. CEI has "social" elements in the sense of the system having certain mechanisms that we find in social technologies, such as the opportunity to see what others have reported, and adding comments to observations.

The CEI application is written for Android, a modern OS where applications are constructed from relatively loose-coupled components. These components are modular units applicable also by other applications, which enable integration with already existing applications and the OS. Programming applications for Android, and smartphones in general, are in many ways rather different than regular programming for computers, especially concerning uncertainty associated with the different resources and the loose couplings mentioned above. Applications in Android are written in the programming language Java, and Android has support for most of the existing regular Java-libraries. In addition, Android has a lot of its own libraries.

The application is part of a demonstration system. It has been tested by the Home Guard on several occasions, most recently at "Øvelse Hovedstad" in September 2013. The application has interesting functionality which can be useful for different parts of the Norwegian Armed Forces, especially to demonstrate the usefulness and the possibilities of mobile technology.

The work is highly relevant for this thesis, specifically due to its similarity in functionality and scope. It is an application that is primarily developed for military use, but can also easily be used for civil purposes. The main functionality is to let users share their positions and observations to a main server. It is also implemented for handheld devices, and in this case, for the Android OS. The application makes use of civil and familiar technologies, and is as mentioned designed for web, smartphones and tablets. All of these aspects show few differences between the CEI app and the app developed in this thesis. However, the major and most important distinction between the apps, is that the CEI one is written in native Android Java code, and is *only* targeted at Android devices. This differs from our app, because of the new focus on platform independent development.

Our app is implemented with web technologies and using a cross-platform framework to provide the necessary tools and Application Programming Interfaces (APIs). An API is a software library that specifies how some software components should interact with each other, i.e., it includes specifications for data structures, object classes, routines and variables. The app in our case is targeted not only at one OS, but up to several, while still writing in one programming language. The focus in our thesis is on both Android *and* iOS, but in theory it could be even more than these two platforms. Apart from the experience and lessons learned by using the CEI system, we also make use of their server. However, we do *not* use other parts of their system, like their application. The CEI solution, and especially the server, is relevant to look at since it has been used in experiments by reporting positions and observations earlier. Therefore it makes sense to use it as an existing, though proprietary, server to connect to.

The next section provides the requirements analysis of the thesis, and defines the requirements that should be fulfilled for our prototype solution to be a success.

## 2.3 Requirements Analysis

In this section, the requirements analysis is provided. We discuss the various requirements we have, and why they are necessary for this thesis. All requirements are summarized in table 2.1 at the bottom of the section. The left column only defines the number of the requirement, for making later references to the table easier. The middle column contains the actual requirement. Lastly, the column to the right defines the priority level of the relevant requirement. This ranges from one to three, where one is the highest priority and thus refers to the most important ones. These requirements are the most important ones to be met.

The first three requirements defined in our table are derived from section 1.3 and are all actually premises for this thesis. They state that the application must be platform independent, that it must at least support the iOS and Android smartphone OS's and that the framework used must be free and open source. To comply with these premises, section 2.6 offers a further discussion and evaluation of possible suitable frameworks. All the premises are fulfilled by design and have a top priority. Since they are allocated priority level one, these are requirements that *must* be met.

One very essential requirement in this thesis, is that our prototype application must work in, and support, DIL environments (also referred to as the tactical edge). These concepts are defined in section 1.1, but easily put they are environments (networks) with possible communication disruptions, bad connectivity and limitation problems. This requirement is important due to the fact that we want our application to work in these types of environments. When PISA encounters this type of network, it should still work and in the best possible manner. It is listed as requirement number four in table 2.1 with a priority level equal to one, i.e., the highest priority. This is the requirement we perform the majority of our testing on in chapter 4, along with the next two requirements.

Another important factor is that the application needs to support both loss tolerant messages that can be lost during transmission, and messages that do not tolerate loss and must always arrive at their destination. These are two additional important requirements we have in this thesis. Since we operate with different kinds of messages, we decided that some of them are more important than others. The ones that do *not* tolerate loss must be taken care of if some connectivity issues arise. Observations and on demand positions are examples of these messages. The messages that do tolerate loss, need only to be counted as successful or failed messages. If one of these messages cannot be properly sent due to loss of connection, they will in fact be lost. Periodic positions belong to this data group. The loss tolerant messages, and the messages that must always arrive are assigned as requirement number five and six respectively in table 2.1. These two also have the highest priority level. Testing is also performed on these two requirements in chapter 4, and they are equally significant as the previous requirement.

REST is used as the communication method in our prototype app, instead of the SOAP protocol. SOAP is more commonly used in military systems, however REST is better suited for use by our solution since we focus on the use of COTS technology in a military setting. Also, SOAP is not natively supported on Android, for instance. The discussion about why REST is preferred over SOAP in this thesis can be found in section 1.1. Section 2.4 provides differences and similarities between REST and SOAP, and detailed information about REST is provided in section 2.5. That the application must use REST as the method of communication is another requirement, requirement number seven in table 2.1 to be exact. It is given priority level one. The requirement is fulfilled by design, since we implement REST in our prototype, PISA.

The application also needs to have a simple and intuitive Graphical User Interface (GUI). GUI is a type of interface that allows users to interact with devices through visual components. For example, the GUI of our application is composed of the colors, buttons, labels, input fields, text and all that is visible to the user. GUI is perceived as the opposite of Command-Line Interfaces (CLIs) where the user is required to type commands on the keyboard for actions to happen. Since PISA is supposed to be a basic reporting app that is easy to use, the requirement of it both being simple and intuitive is essential. The application needs to be simple to use in relation to both intuitiveness and complexity. It should be easy to understand what to do, how to do it and what functionality PISA offers. Additionally, the offered functionality cannot be too complex or hard to perform. For instance, big buttons are preferred since it needs to support "fat fingers" when users are out in the terrain walking or running, and possibly wearing gloves. The GUI requirement can be viewed in table 2.1 as requirement number eight. It is given a slightly less priority than the others, namely priority level 2. This is because the GUI and its testing is not prioritized and carried out as extensively as the previously described requirements concerning functionality. Priority level 2 consists of requirements that *should* be met. The GUI is evaluated by a small group of test subjects, and all the information about it can be found in section 4.6.

Requirement number nine in table 2.1 is about using standards when possible. We want to facilitate for further interoperability in the best manner, therefore using standards is a good way of possibly achieving this. For instance, we use a standard data format, Extensible Markup Language (XML), for saving positions and observations on the server we implement. Further, we also use a specific format for positions, which is a NATO standard. However, other formats are also used in our implementation due to the integration with the CEI server. The requirement of using standards when possible is given priority level three, which consists of requirements that are not that essential. The requirements with this level *can* be met. The requirement is fulfilled by design, because we implement the standard formats in our prototype application.

Facilitating interoperability towards back-end infrastructures like NII and SOAP Web services is also important. This is desired so that future coupling with our prototype and NATO standard infrastructures is possible. This requirement has a close correlation with the previous requirement, since the presence of standards enables the probability of this requirement to be satisfied. We use NATO interoperable formats in our solution, which opens for potential later connection with NII and SOAP Web services infrastructures. This requirement is the last one, number ten, in table 2.1. It also has the lowest priority, i.e., level three. This requirement concerning facilitating interoperability towards NII is also fulfilled by design, since we use the appropriate formats which enables this opportunity.

Finally, an aspect that we have not addressed is optimization of battery life on smart devices. This is not a requirement in this thesis, but it should be noted that battery life on handheld devices is an important factor. Experiments with CEI showed that battery life was an issue with long operations out in the field. With high or intensive usage on the device, the battery could be depleted within a relatively short period. When out in the field reporting observations and positions, the possibility of charging your device can be very limited. Therefore, ways of optimizing battery life are quite essential in cases like these. We have not included a requirement concerning this aspect due to time restrictions, and that options of optimizing battery life are limited when it comes to using an application level framework like we do. One usually must have access to lower layers to be able to modify features that substantially affect battery life.

## 2.4 SOAP vs. REST

SOAP is the messaging protocol much used in military systems in general. It is meant to be applied towards back-end infrastructure like NATO's NII. REST is used more towards the "edge" of the infrastructure with cheap COTS solutions. The key characteristics of both SOAP and REST Web services are summarized in table 2.2, which is derived from [9].

We want to use REST since it is the most suitable of the two for our purposes. We only use HTTP/HTTPS for transportation of the messages between PISA and the servers. REST might not be a full standard, but it is widely adopted in a huge amount of civil applications. REST is not restricted to use only XML, as SOAP bases its messages on. REST can be used with either XML [33], JavaScript Object Notation (JSON) [26], Atom [25] or some other data format of your choosing. SOAP is the foundation of a complete middleware, and can be used for almost any purpose. REST is a good alternative when needing only simplistic point-to-point connections. SOAP is identified as the key enabler for realizing NNEC, which is important in military systems. Luckily, there is some interest shown in using REST for certain applications for NATO. That is why we research the opportunities of utilizing REST instead of SOAP in our type of application.

The next section contains more information about the used communication method in our thesis, namely REST.

| # | **Requirements** | **Priority** |
|---|---|---|
| 1 | *Premise - Application must be platform independent* | 1 |
| 2 | *Premise - Application must support major smart device OS's (iOS and Android)* | 1 |
| 3 | *Premise - Application must be implemented using a free and open source framework* | 1 |
| 4 | Application must support, and work in, DIL environments (e.g., tactical edge) | 1 |
| 5 | Application must support loss tolerant messages, i.e., positions (these messages can be lost) | 1 |
| 6 | Application must support messages that are not tolerant of loss, i.e., observations (these messages must arrive) | 1 |
| 7 | Application must use REST as communication method | 1 |
| 8 | Application needs to have a simple and intuitive Graphical User Interface (GUI) | 2 |
| 9 | Use standards when possible | 3 |
| 10 | Facilitate interoperability towards back-end infrastructures (SOAP Web services, NII, etc.) | 3 |

Table 2.1: Requirements and Premises

| **SOAP** | **REST** |
|---|---|
| Can use almost any transport | Uses HTTP/HTTPS exclusively |
| Somewhat complex | Very easy to understand |
| Industry standard | Lacking in standardization |
| Based on XML | Can use XML, JSON, etc. |
| The foundation of a complete middleware | Good for simplistic point-to-point connections |
| Identified as the key NNEC enabler | Some interest in NATO for certain applications |

Table 2.2: SOAP Web services vs. RESTful Web services

## 2.5 REST

REST is an architectural style for designing networked applications. Simply put, it is an architectural pattern to transmit data over a connection between a client and a server. This style is also applied to the development of Web services as an alternative to other distributed-computing specifications like SOAP. These REST Web services are often referred to as RESTful Web services.

The communication takes place in a request/response messaging pattern. REST has no definition of the publish/subscribe pattern, which SOAP can be used for. This means that REST utilizes only point-to-point communications. Uniform Resource Identifiers (URI's) are employed to identify the different resources, and the location of where to find them. REST is also stateless, which means that the necessary state to handle the request is contained within the request itself. In addition, REST is format independent. It can use JSON, XML, Atom, images and more as the specified data format. Standard HTTP methods are used to post data (POST and/or PUT), read data (GET) and delete data (DELETE).

REST uses a uniform interface, and this is provided by HTTP [24] and HTTPS. REST relies on a stateless, client-server, cacheable communications protocol, and in virtually all cases, the HTTP protocol is used. The idea is that, rather than using complex mechanisms like SOAP, Common Object Request Broker Architecture (CORBA) or Remote Procedure Call (RPC) to connect between machines, simple HTTP is used to communicate between client and server with REST. It is regarded as a lightweight alternative to mechanisms like RPC and SOAP Web services. Despite REST being simple, it is fully-featured and can be used in most scenarios as the other mechanisms can.

## 2.6 Overview of application frameworks

Based on our previous research [45], we have provided a summary of the relevant frameworks that support cross platform development features below. The work in [45] was aimed at three different platforms, namely iOS, Android and Windows Phone 7 (WP7). The app that was developed was meant to be an app where you could search for job vacancies.

Either way, the earlier work is still very applicable in our scenario. Following is a list of the seven appropriate frameworks found in the survey from [45]. These were selected after the first screening, where the requirement of support for all platforms (iOS, Android and WP7) was most essential:

- PhoneGap (Cordova)
- Kony
- Rhodes (RhoMobile)
- FeedHenry
- webMethods Mobile Designer (Software AG)
- Mono (Xamarin)
- Mowbly (CloudPact)

All the previously listed frameworks are described in table 2.3 on a high level, yet providing information such as chosen technology, type of license needed, prevalence in communities, reviews, etc.

Based on table 2.3 we find that out of all seven candidates, PhoneGap is the "overall winner" mainly due to its license. Being completely free and open source under the Apache License, Version 2, PhoneGap really is the only framework that meets all requirements in section 1.3. Also, PhoneGap is preferred because of the positive experience using it in [45]. PhoneGap also proves to be a very mature and acclaimed framework, with support for both iOS and Android, and having such a large community. It should be noted that whenever "Web technologies" is mentioned in table 2.3, we specifically mean HyperText Markup Language (HTML) version 5 (HTML5) [34], Cascading Style Sheets (CSS) [32] and JavaScript [19].

A detailed description of PhoneGap is listed below, with specific information about the framework's features, API, support, community, etc.

### 2.6.1 PhoneGap

PhoneGap [14] is an open source framework for building cross-platform mobile apps using standards-based web technologies, i.e., HTML5, CSS and JavaScript. Instead of using different languages and frameworks, PhoneGap offers these web technologies to bridge web applications and mobile devices. PhoneGap is widely used, with over 1 million downloads and 400,000 active developers. Thousands of PhoneGap-developed apps are available in mobile app stores and directories. The different apps can be explored on their official PhoneGap site [14].

| Framework | Technology | Availability | Maturity | Cost |
|---|---|---|---|---|
| **Rhodes** [30] by Motorola | Web technologies and Ruby | Easy to download, uses Eclipse as Integrated Development Environment (IDE) | Well-known, good number of users, growing in popularity, fairly frequent updates, mostly good reviews | Free to download, but has publishing costs |
| **PhoneGap** [14] by Nitobi, owned by Adobe Systems | Web technologies | Very easy to download, uses Xcode, Visual Studio, Netbeans or Eclipse as IDE | Very well-known, many users (maybe most of the reviewed frameworks), frequent updates, excellent reviews | Completely free and open source |
| **Kony** [29] by Kony | Web technologies | Not easy to download (must contact people behind it, very little information), based on Eclipse, but has own IDE, called KonyOne | Not very known in communities, several major partners (IBM, Microsoft, etc.), little information on updates, some reviews from homepage, but few on Google | Not free, commercial product |
| **Mono** [36] by Xamarin | C# for all logic, User Interface (UI) left to each platform | Somewhat easy to download, uses Visual Studio and has own IDE called MonoTouch | Known especially on GitHub and .NET webpages, many users (maybe most of the reviewed frameworks), periodic updates, good reviews | Not free, needs one license per product |
| **FeedHenry** [20] by FeedHenry | Web technologies | Web based IDE with simple registration, plugin for using Eclipse, but must go through the cloud to publish, simple for individual developers, but generally hard to find information | Very little known, not mentioned in many forums at all, quite possibly few users, relatively periodic updates, nearly no reviews | Not free, commercial product |
| **webMethods** [31] by Software AG | Basically only Java, pushes to C# on demand | Pretty hard to acquire, has only plugin for Eclipse, needs licenses, but close to no information about it (little information in general) | Software AG is huge, so one can assume webMethods is also used, probably designed for commercial use, little information on updates, nearly no reviews | Not free, commercial product |
| **Mowbly** [2] by CloudPact | Web technologies | No download available (at that time), can only use a web IDE, offers different licenses | New framework, not many users (at least not yet), not much information on updates, no reviews to be found | Not free, commercial product |

Table 2.3: Reviewed frameworks (summarized from [45])

The software underlying PhoneGap is Apache Cordova. So, the names PhoneGap and Cordova are used interchangeably in communities and even official documentation. However, they both refer to PhoneGap, which is the official name. As stated, PhoneGap uses web technologies instead of device-specific languages such as Objective-C, thus it is regarded as a hybrid framework. This means that it is neither truly native (since all layout rendering is done via web views instead of the platform's native UI framework) nor purely web-based (because it does not produce web-apps, but package them as apps for distribution and gives access to native device APIs).

The tests we conducted in [45] showed that PhoneGap was the only framework that provided 100% interoperability between the different mobile platforms, where you could use the same code everywhere with little need of tweaks. The others struggled to support the same features properly for all three platforms. In addition, PhoneGap is the framework with the largest community, most support, easy to set up and use, and has the best reviews throughout. It is also completely free and open source. PhoneGap proved to be very stable, and supports most of the featured API.

To achieve the goal of this thesis, which is described in section 1.4, the framework must support API and features like the camera and Global Positioning System (GPS). PhoneGap has support for these two (respectively named Geolocation and Camera), and many more. See table 2.4 for an overview of necessary API that is supported by the different platforms with PhoneGap.

For more information on PhoneGap, for instance the API support, guides, docs and tutorials (e.g. Wiki- and Google Groups pages), see PhoneGap's homepage [14]. PhoneGap is at the time of writing available in version 3.5. However, the prototype is implemented with version 3.3. This is due to it being the latest version when the work on this thesis started.

Additionally, if you are interested in information about how to install and set up a simple project with PhoneGap, read appendix A for a summarized and simple step by step tutorial.

## 2.7 Summary

This chapter essentially featured the background and requirements analysis of the thesis. Some relevant work was included to provide some insight into what has been done in earlier and ongoing related projects. The CEI prototype system received extra attention and was introduced in this chapter. We are going to use the server component of CEI as an additional server to communicate with, to prove that interoperability between our

| **Feature** | **Description** | **iOS (3GS+)** | **Android** |
|---|---|---|---|
| Camera | Provides access to device's default camera application. Can take photo using the camera, or retrieve one from device's image gallery. | ✓ | ✓ |
| Capture | Provides access to device's image, audio and video capture capabilities. Can capture images using device's camera application and return information about captured image files. | ✓ | ✓ |
| Geolocation | Provides access to location data based on device's GPS sensor, or inferred from network signals. Can return or watch for changes on device's current position. | ✓ | ✓ |
| File | Providing functionality to read, write and navigate file system hierarchies. Can upload and download files to and from a server. | ✓ | ✓ |
| Connection | Provides functionality for detecting any available type of network, and provides access to the connection information. | ✓ | ✓ |
| Notification | Providing access to visual, audible and tactile device notifications. An alert, sound and vibration are examples of notifications. | ✓ | ✓ |
| Storage | Provides functionality to store data locally on the device. Utilizes HTML5's alternatives for storage: LocalStorage, WebSQL and IndexedDB. | ✓ | ✓ |

Table 2.4: PhoneGap - overview of supported features (adapted from [13] and [12])

solution and an existing solution is feasible.

The requirements for this thesis were discussed, and the ones applicable to our scope are defined in table 2.1. This table also includes the premises of our thesis, which is defined in section 1.3. The rest of the requirements concerned what type of messages we wanted to support, what data tolerated loss or not, that the application needed to work in DIL environments, GUI had to to be simple, REST was used for communication, and so on. All these requirements must be fulfilled to at least some degree for our prototype solution to be a success.

Discussions around the differences between using SOAP or REST for communication purposes were included. SOAP was widely used in other related projects. However, in this thesis we are going to use REST since it is better suited for our solution. A comparison of REST and SOAP was therefore provided, along with a more thorough description of REST.

Lastly, our application is set to use a platform independent framework since we want to implement our solution both for the iOS and Android mobile platforms. An overview of suitable candidates then followed, with a decision on which of the frameworks were the most appropriate for our objective being taken. PISA is therefore going to be designed and implemented using the platform independent PhoneGap framework.

The first three requirements defined in table 2.1, the premises of our thesis, is fulfilled once the platform independent application is implemented, and since the application supports major OS's (iOS and Android) and the application is developed with a free framework. These are all covered since we are using the platform independent PhoneGap framework which is free and open source, and supports both the iOS and Android platforms (and several others).

Both the design and implementation of our prototype solution are provided in the next chapter.

# Chapter 3

# Design and Implementation

This chapter introduces the design and implementation details of our prototype application, PISA, including our basic server. It also explains how the communication with our server, and the one provided by CEI, takes place. What type of data and how it is sent between PISA and the servers is described in detail. As well as the application logic design, the GUI design is presented and discussed. All essential design choices are mentioned and justified in the sections below. First, the overall design is presented. The general design of the application is then introduced, before the specific design of both the application and the server are described. The implementation of PISA and all of its major components, along with following code listings is described to further demonstrate the implementation specifics. Additionally, the implementation of our wrapper server is explained.

## 3.1 Overall Design

The application implemented in this thesis can choose to transmit sensor data to a server on demand, or do it on a periodic basis. With sensor data, we mean GPS-assisted information such as the location of the user, along with the camera functionality. The app is set to work on both Android- and iOS devices, and there are no limitations as to how many instances of the app that might exist.

Figure 3.1 depicts an overall and simple design of how the application works. The application can be implemented once, then deployed on several iOS- and Android platforms that may be in communication with the central server, thus having the possibility of a many-to-one communication. It is also worth knowing that the central server in this thesis, is actually divided into two entities; Our own basic server, and the CEI server. CEI was introduced in section 2.2, and appendix B discusses the connection details of the CEI server. The reason for connecting with two different servers is the implementation focus, and the fact that PISA is aimed to work with existing servers other than our own, like CEI. So in reality, this gives our solution the ability of having a many-to-many communication. All this, however, is further explained in the next section.

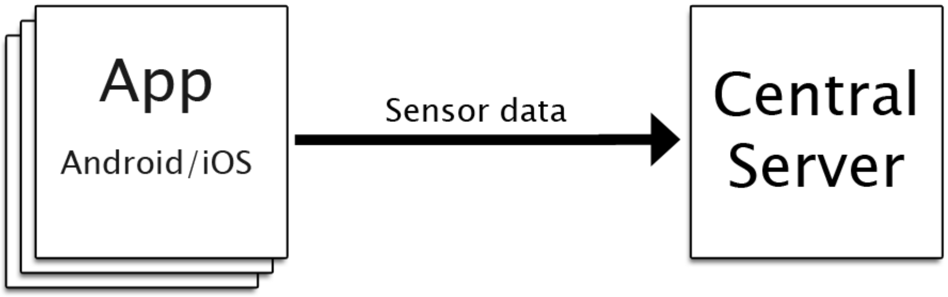


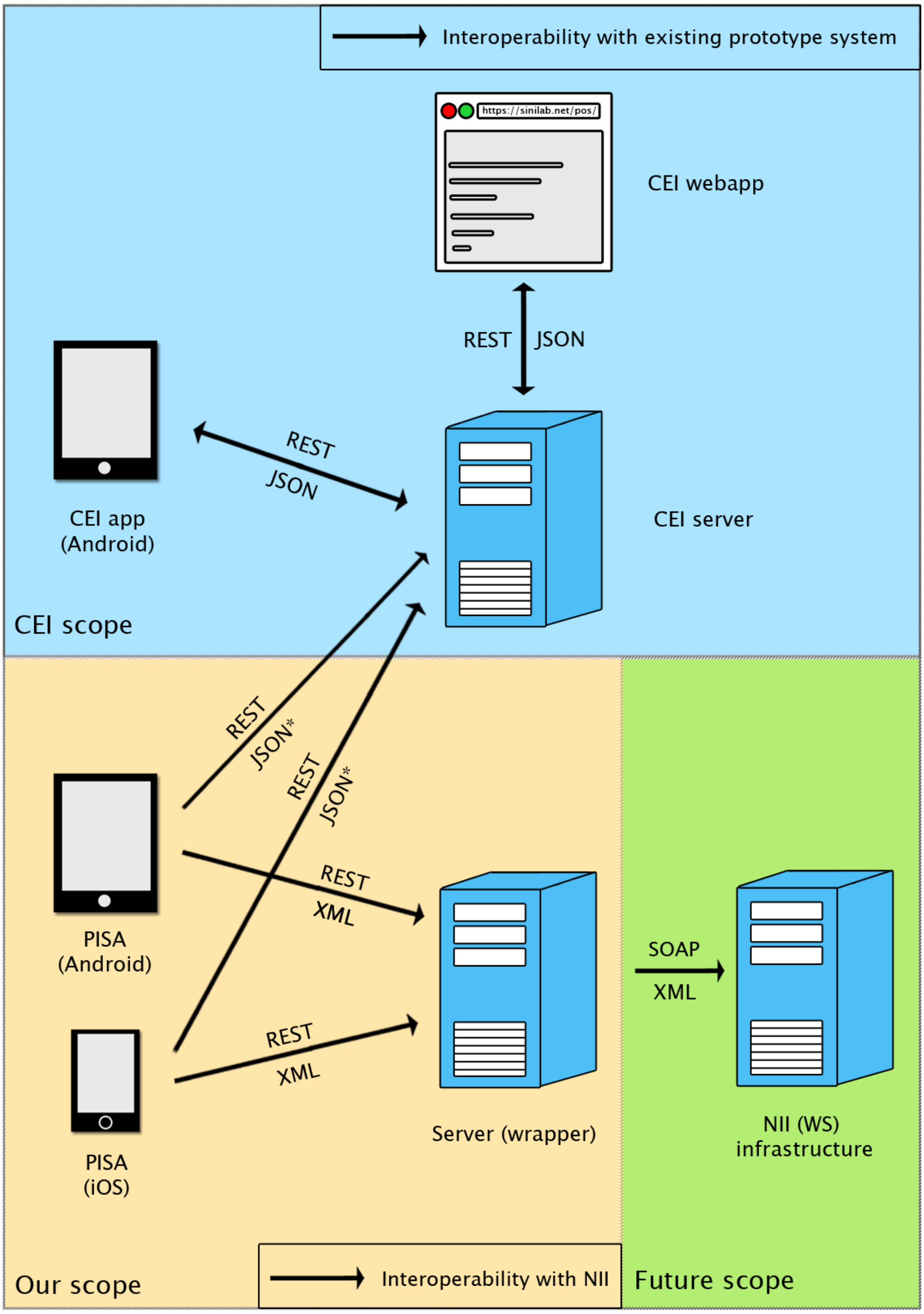
Figure 3.1: Overall Design

Implementing a complete server is beyond the scope of this thesis. Therefore, we also connect to an already existing one (CEI). Furthermore, we also create a very basic server (wrapper) to facilitate fulfilling the requirement of interoperability with NII. The wrapper server is responsible for receiving sensor data, listening for incoming messages, storing the sensor data along with information on the current transmitter on a file, and replying back to the transmitters.

## 3.2 Design of Solution

A more detailed look at the design and scope of our application can be seen in figure 3.2. It illustrates how PISA communicates and works with the other components of our solution. The figure is split into three separate areas, which define this thesis' scope, CEI's scope and the scope left for future work. These distinct zones are included to explicitly show the scope that belongs to our solution, what already exists, and what is left to be implemented by others.

The bottom left area of the figure defines what is developed by the author in this thesis. This is our scope. The area on the bottom right is the future scope. It illustrates what is left for future work to make our solution completely compatible with NII, or a Web services infrastructure. This is a necessary component to make PISA work with



* Position and observation requests are sent with JSON, with the exception of observations with an image attached. These are sent as a form submit with encoding type "multipart/form-data" to allow sending of entire files in the data.

Figure 3.2: Detailed Overall Design

other NATO standard solutions. Our server only works as a wrapper (also often called a proxy) server. This is due to the scope of the thesis, which is found in section 1.4. However, the server contains standards compliant XML files, which makes the work of interoperability with NII far less extensive. Implementing this, however, is left for future work.

The topmost area of the illustration shows the components developed by CEI, which are integrated with our solution. It includes its Android application, web application and most importantly, the server. In theory, we only interact with the CEI server, which the unidirectional arrows indicate. In practice, however, we also use its web application for login purposes. The CEI application is not used in the thesis. It is only provided in the figure to show the communication between its app and server. This area, and the arrows leading to the server from PISA, are included to illustrate that our solution also supports interoperability with other existing prototype systems, such as CEI.

All communication between PISA, and the wrapper server and CEI server is transmitted through the REST interface. This requirement is defined as number seven in table 2.1, where REST used as communication method is mandatory. The communication from the CEI application and its server also uses REST. The only deviation is for interoperability with NII, the messages must be mediated with the SOAP protocol. Regardless, this is beyond our scope. The data format used between PISA and our wrapper server is XML, specifically the NATO Friendly Force Information (NFFI) format and the FFI-incident format. See [38] for an overview of NFFI, and STANAG 5527 for the complete specification. NFFI is a NATO compliant standard, and it is used for tracking the location of *blue forces*, or friendly forces. By utilizing NFFI, we further help fulfilling requirement number ten in table 2.1 by design on the format level. The other format, FFI-incident, is not a standard. It is a proprietary format for reporting observations, but it is built with XML, which is a widely adopted standard. However, the reason why we do not use a standard here, is because there does not exist a format for observations in NATO that is agreed upon. Since NFFI and XML are used, we further help requirement number nine in table 2.1 being fulfilled by design. XML can also easily be used with SOAP. In [11], you can find an appendix describing the complete FFI-incident format. The two formats and their attributes are further described in section 3.3.7.

CEI uses JSON as the chosen data format for communication between application and server. Only completely proprietary formats are used for reporting positions and observations in this solution. JSON is an open standard that uses human-readable text to transmit data objects over a network. However, by using JSON and proprietary formats, conflicts arise with our requirements. Requirement number nine in table 2.1 is then compromised a bit. Unlike XML, JSON cannot be used with SOAP. So, future interoperability with NII therefore becomes somewhat complicated.

Additionally, PISA uses JSON as data format when communicating with the CEI server. This is the required data format for its server. As seen in figure 3.2, the arrows from our PISA devices are labeled "JSON" messages followed by an asterisk (*). The fact is that not all messages are sent as JSON data. Positions, and observations with position and description only, are always submitted as JSON objects. Observations including an attached image are posted as a web form submission. This other data format is needed because of the image being sent as a file, and not as a textual string representation of the image, as it does with FFI-incident. Technical details on this event can be found in section 3.3.6.

The next two sections further describe the functionality and design of the application, and also the server component of this thesis.

## 3.3 Application Functionality

This is where the logic and functionality of our prototype application is discussed, and we argue what the app does and why. We also provide information on that there are other alternative ways of performing the functionality than we do. However, we try to justify why we choose the way of implementation as we do. All the opportunities we had are mentioned and explained. The subsequent sections describe PISA's general functionality along with design choices and possibilities, before the main functionality divided into their own section is discussed, e.g., one for on demand position, one for periodic position, one for observation with description, etc. The last section describes how the configuration part of PISA is designed, and explains the different attributes used in this application.

PISA supports transmission of positions and observations as the main functionality in this thesis. This is because PISA is a basic reporting application. PISA is an application that has the possibility of making information from the sensors a device has available, and we have decided to focus on the GPS and camera sensors. We focus on these because they are common on the majority of mobile platforms, and because they offer the most benefit in our case. GPS simplifies position reporting, and camera support allows for sending of observations with images, and not just textual descriptions.

### 3.3.1 General

In this section, several design choices are discussed. General functionality is also described here. They are presented in this separate section because they match multiple of the messages defined below.

Due to our application occasionally being used in DIL environments, we have decided to utilize the caching feature. This is provided by the PhoneGap framework, and more specifically the underlying WebView implementations of something known as *local storage* or *web storage* [35]. We use caching (local storage) to store our essential data on the device when the messages fail being sent to and received at the destination point, i.e. the servers. If the connection is lost before or while the message is sent, the user is notified and the message is stored for later retransmission. If PISA is used in a limited network, and the message takes too long being successfully sent, the user is notified that a communication error occurred and the message is cached. This is a timeout configured in our application, so the user does not have to wait for a very long time before he is aware of what is going on.

The user needs to explicitly send the latest position and observation if the previous one failed. This could be done automatically, but it is not. This is due to the fact that this application might be used in limited networks (DIL environments, tactical edge, etc.). To flood the network with retransmissions of messages is therefore not optimal. By forcing the user to do this himself, it saves network resources, which could be scarce to begin with.

Only one position and one observation is stored in the cache at a time. It is always the first position or observation the user wanted to send, but was unable to due to connection loss, that is cached. This is done to limit memory usage on the device. By caching too much information at a time, the device could crash. It is always the first position/observation that is stored, since it initially was this information the user tried to send. It could however easily been the last position or observation also. This is not done because we wanted to keep the *initial* message.

How the application is now, the user must himself detect that the network connection is back up after losing it. PISA should itself detect that the network is back again, at least after a device-side connection loss occurred. It can use this information to remind the user that there exists data that is not yet sent. Similarly, if you suddenly succeed with sending for example periodic position, the user should then be reminded that there are other existing data in the cache.

Timeouts are defined in PISA to let the user not wait for too long, when messages take a long time to send. These timeouts are statically set in advance, and the user has no way of adjusting these timeouts. There are of course other ways of doing this. The application timeout could be dropped, or one could possibly adjust the timeout according to the specific network type the device is situated in. Another possible way, could be to find a way to synchronize the app timeout with the underlying transport protocol timeout. These opportunities are not researched in this thesis due to time restrictions, and we think that our timeouts are suitable for coping with most scenarios.

This issue has a direct correlation with the one defined above. The inaccurate timeouts could also lead to duplicate messages on our server. Management of duplicates either by PISA or the server should be implemented. Unfortunately, this is left for future work, due to time constraints.

### 3.3.2 Position - On Demand

The on demand positions are the positions the user manually retrieves and sends to the servers. These are sent one at a time, and only when the user decides to send one by pushing the send button. They can be sent to both the wrapper and the CEI server, and the content of these two messages is different. These positions are part of the messages that cannot be lost, they must always arrive at the server at some point. We have on demand positions as part of our main functionality because they are important if for example periodic positioning is unavailable. If a user decides that he is on an essential location at a specific moment, he may explicitly send this location. These positions are sent using two different formats. NFFI is the format of the messages sent to our server, and the data format used for sending to CEI is JSON. NFFI is a NATO standard, and consists of XML data. JSON is another data format, and the specific format sent to CEI is proprietary. NFFI is applied since we want to facilitate for interoperability with NII, and JSON is used because it simply is dictated by CEI. Figure 3.3 illustrates the message contents of the on demand positions.

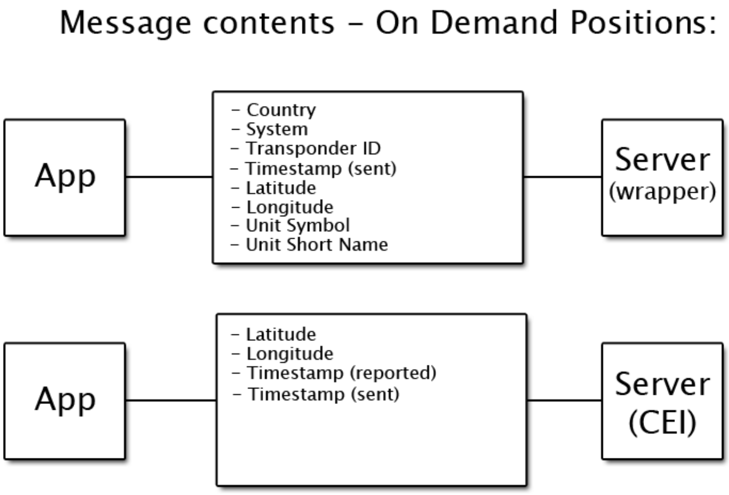


Figure 3.3: Message Contents of On Demand Positions

The on demand position sent to the wrapper server is composed of the NFFI attributes (country, system, transponder ID, unit symbol and unit short name), along with the latitude and longitude of the user's location. The NFFI attributes are described in section 3.3.7, including the rest of the configuration. The latitude and longitude is the data that together create the position of a user. A timestamp is also included on the format `YYYYMMDDHHmmSS`, which provides the current date (in year, month and date) and time (in hours, minutes and seconds). This is the timestamp of when the position is successfully sent to the server. If the position fails to be sent however (due to a connection error), the timestamp is updated the next time the user resends the position. The timestamp updates itself to contain the time it is successfully sent, and not first registered. This is another possibility. The timestamp could instead have been the timestamp of when the actual position was initially registered. This alternative is not chosen in this thesis, because we want to be able to see when the positions are successfully received on the server side. This is really a matter of personal preferences, but both ways are relevant. We were limited to only one timestamp, due to the NFFI format, or else we could have used both.

The on demand position transmitted to the CEI server is different from the one sent to the wrapper server. It does not contain the NFFI attributes, since the format used now is a proprietary position format built with JSON. The message only has the essential parts, i.e. the latitude and longitude. In addition, two different timestamps are included here. One timestamp is for when the message is successfully sent, like with the NFFI message. The other timestamp is for when the position is first reported. These two may be identical, if the position is successfully sent straight away. However, if connection issues arise, the two timestamps will differ. The reported timestamp is set once the user first tries to send the position. If it fails, it remains the same until it is finally sent. However, the sent timestamp updates each time the user tries to send the position.

### 3.3.3 Position - Periodic

The periodic positions are the other option to the positioning functionality. The user must manually turn on periodic positions, and once this is done PISA automatically retrieves and sends these positions to the wrapper server without any further user involvement. The interval of these periodic updates is configurable as a user setting in the PISA. Like for on demand positions, these could be sent to both of the servers. However, due to time limitations of the thesis it is only possible to send these messages to our server, and not CEI's. Regardless, the functionality is working, so it proves that it can be done. NFFI is still the format used for these messages. These positions are the only member of the data group that tolerates loss. It is fine to lose these messages. The periodic positions are employed in this thesis because they free the user to do other things, given that the network is usable. The content of this message can be found in figure 3.4.

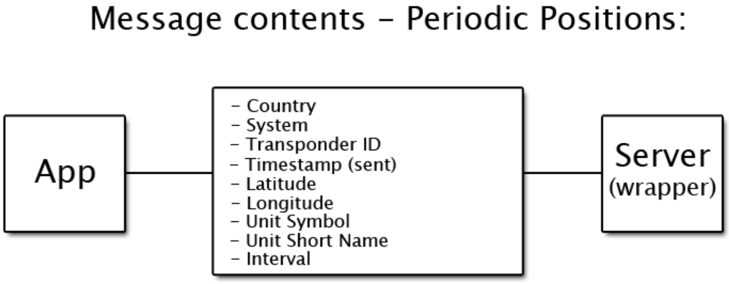


Figure 3.4: Message Contents of Periodic Positions

To mention it, the periodic position message to CEI would incidentally have been the exact same as with on demand positions. The periodic positions with the NFFI format are also very similar to the on demand positions. Only one additional attribute has been added, namely the `interval`. The interval defines the number of seconds the periodic positions should be sent between each other. If ten seconds is chosen by the user, the positions are sent every tenth second. The rest of the message content is exactly the same as with the NFFI format for on demand positions. With the addition of the extra interval field, the data exchanged between PISA and our wrapper server is not directly compatible with the NFFI data format. Interoperability with the standard NFFI data format is easily regained by having our wrapper server remove this extra information before providing it to others. This is because we choose to include the periodic interval as an extra field in the messages, and it is done to distinguish (on the server side) which message is from an on demand position, and which is from a periodic position. We can easily drop this, so further interoperability is not compromised. We choose to do it this way, since it is unproblematic to remove it again. There are other methods of distinguishing on demand and periodic positions, like sending the different types of positions to various endpoints (ports) on the server. However, this would have led to more work on the server side, which is not the main focus of this thesis.

As we said earlier, periodic positions can be lost, and they are not at any point cached on the device. Another possibility could be to preserve all of them. This way, we could achieve a history feature. The history of the periodic positions could be used afterwards to track and display the movement of the user, for example in a map. We do not include this functionality, because we have a greater focus on leveraging the network and its capabilities, than implementing functionality that can be used for analytical purposes.

PISA only keeps track of the number of periodic positions that are tried sent to the server while periodic positioning is turned on. Once the periodic positioning is turned off again, the user is displayed the number of both successful and failed periodic positions. If the network connection is lost during this time, information about this is of course provided to the user. PISA could obviously keep track of more information than just the number of sent positions. It could have kept information such as timestamps, the time it took to send, etc. However, we wanted to have as little information as necessary, to try not to overwhelm the user with too much text and information.

### 3.3.4 Observation - Description

Observations are the other essential component of the main functionality in this thesis. We want to include the possibility of sending observations in our application, because in a handheld reporting app, observations are a vital component. For users to be able to report observations along with positions when out in the field is very important, since they together define basic reporting functionality.

The observations are divided into three different combinations of sending a specific type of observation. The user can either send a description alone, a description and a position or a description, position and an image. We separate them since we do not want to put too strict limitations on the user compared to what information they have to fill in. As the user may often be in a stressful situation, it is more important that the user can enter and send the information the user considers important, and that the user is not be prevented of doing this if he for instance has no GPS coverage (e.g., if he is indoors), than that we get the sent information as complete as possible.

This section describes the first of the three combinations, namely the observation containing a description alone. We allow these three particular combinations since we think they are a representative selection of the possible combinations. A description alone could be essential if the user had no GPS coverage. A description and a position covers the essentials of reporting an observation. An observation including a description, position and an image is the complete version where all aspects of an observation are covered. There are four other potential combinations: Position alone, image alone, image and description and lastly, image and position. A position alone is not an observation, but only a position, which we already have separate functionality for. An image alone requires more resources to send than a description alone, and we also have the possibility for a user to take a picture and not send it immediately, but later, since the images are stored on the camera roll once taken. An image and a description could be possible, but then the position of the observation would be missing. We have one observation type without position (with description only), and we anticipate that one alternative is sufficient. An image and a position is more logical to include, however it is omitted due to time restrictions.

This type of observation can only be sent to the wrapper server, since the CEI server requires at least a latitude and longitude. This scenario is included in accordance with the supervisors, since the possibility of sending just a description also is desired. The format of the FFI observations is one called FFI-incident. We use this format since it is a format developed by FFI, and the message is built with XML. Unfortunately, it is a proprietary format, and not a standard. FFI-incident is used due to there being no standard format for this that everyone agrees on in NATO. Additionally, it is widely used and tested in applications used by FFI. Figure 3.5 illustrates the contents of the observations containing a description to our server only.

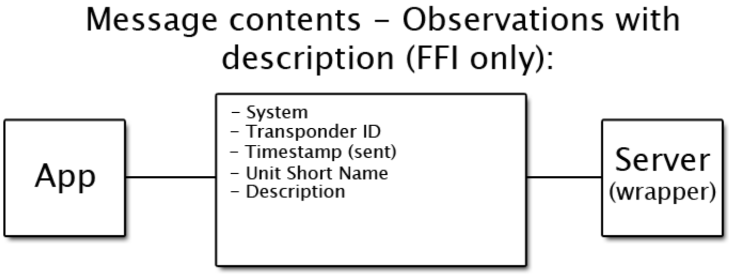


Figure 3.5: Message Contents of Observations with Description

This type of observation can only be sent to our own server, and not CEI's. This is due to CEI demanding their own specific methods and alternatives of observations. Sending only a description to CEI is not possible. The message to our wrapper server contains three of the five NFFI attributes, namely the system, transponder ID and unit short name. Again, a timestamp of the time the message is sent is also included. The new content in this first type of observation message, is the description. This attribute has the textual description of the incident the user observes.

### 3.3.5 Observation - Description and Position

The second option of the observation messages is the observation containing a description and a position. This kind of observation can be sent to both the wrapper server and the CEI server. FFI-incident is still the format being used with this observation as well. The content of the same observation message to the CEI server, is different however. JSON is used here instead of XML, like with the positions. CEI build most of its messages as a JSON object. It is once again a proprietary format, and no standards being utilized here. The contents of these observation messages can be viewed in figure 3.6.

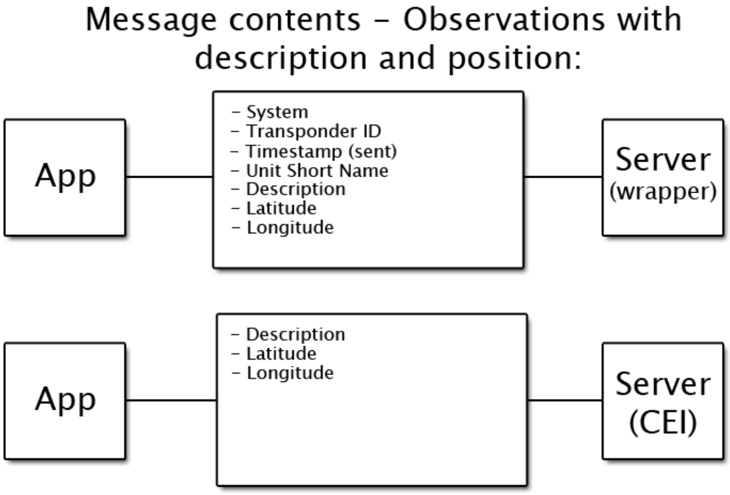


Figure 3.6: Message Contents of Observations with Description and Position

The FFI version of the message contains the same NFFI attributes as the previous type of observation. The timestamp and description are also included. The description is the minimum of what the user must fill in of optional data when sending an observation. In this case, two new fields are added: Latitude and longitude. They represent the position of where the observation took place. All these properties together form the observations with description and position.

The CEI observation message is again slimmer than the FFI option. It consists of the description, latitude and longitude, i.e., only the necessities. A timestamp should be present, but due to integration difficulties from both ours and CEI's side, it was later dropped. This was decided after talking to one of the authors/developers behind CEI [39].

Now, when we also send observations to the CEI server, there is another possible combination we could have utilized. CEI determines their own formats and what the messages must contain, and the CEI server in fact accepts observation messages with a position and image only (without description). However, while we recognize that the position and image observation should be a valid alternative, we choose to not include this message in this thesis. This is only to try to keep the different alternatives to a minimum, and the fact that this information was omitted and not discovered until the time left was insufficient.

### 3.3.6 Observation - Description, Position and Image

The third and last option of the observation messages is the observation having a description, position and image. This message can also be sent to both servers. The message sent to FFI uses the FFI-incident format, and the message going to the CEI server contains serialized web form data, and not a JSON object as with the previous messages to CEI. This must be done since sending these observations including an image as JSON objects seemed impossible. After further discussion with a CEI developer, another solution is to populate a web form, and send these form data with the message. Figure 3.7 illustrates the message contents of observations containing all data, i.e. description, position and image.

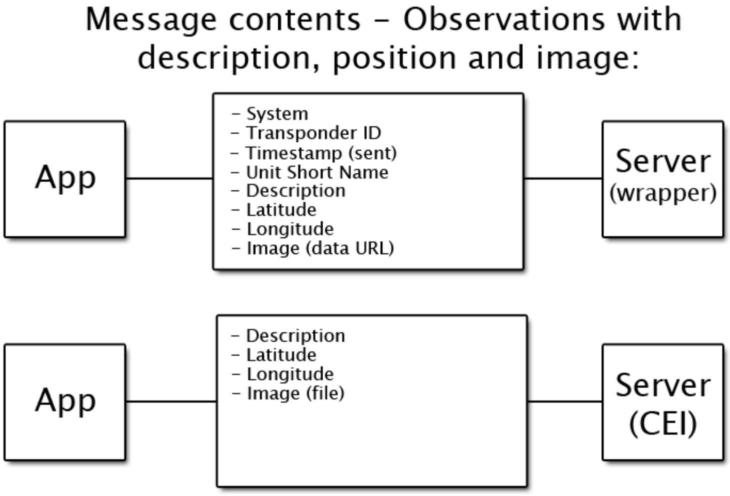


Figure 3.7: Message Contents of Observations with Description, Position and Image

The observation message going to the wrapper server still includes the same NFFI attributes as the two last ones. In addition, the timestamp, description, latitude and longitude are included, i.e. the ones from the previous observation option. What is new here, is the image data. The image attribute is, in the FFI case, represented by a text string, or a data Uniform Resource Locator (URL). A URL identifies and names a resource (our image) using the location information or resource address. This URL contains the base64 encoded text string. The image is encoded with base64 to make it possible to put binary data in XML files. This way we can convert the image to a textual representation of itself, so it can be included as a part of our XML formatted message, which is required by the FFI-incident format.

The message that is sent to the CEI server is implemented in a different way this time.

It is sent as data retrieved from a web form. This observation message consists of the description, latitude, longitude and the image. The timestamp is omitted here too, for the same reasons as with the previous observation type. However, the image data here is picked from a file input. This means that it is represented as a file, and not a base64 encoded string as for the FFI message. Also, since all data is sent in a completely different way this time, the two messages vary quite much.

The handling of images is performed in two separate ways in our application. When choosing to send this type of observation to our own server, you can select between taking an image or picking one from the camera roll. And this is performed with functionality provided by PhoneGap. This is done because we want to have the ability to save our picture automatically after taking a new one. The taken/chosen picture is displayed as a large thumbnail in the application window. When you encounter a loss of connection, the data is stored locally on the device. If the user navigates to another page and then back again, and he tries to send the cached data, the device often crashes due to trying to fetch the big data from the image. A workaround for this, was to save every new image after taking an image of an observation, and storing them on the camera roll/photo library. If the user needs to resend the observation, he may have to reselect the image from the camera roll. This way, the caching issue is avoided.

In addition, if a user only has time to take a picture of an incident, he may later (after sending other observations in the meantime) remember that the previously taken picture was very important, and still send it because the image is saved to the camera roll. There exist other, and possibly easier and more standardized, ways of performing the handling of images. The HTML file input could have been used, but then we would not have gotten the base64 encoded data URL of the image just as easily. Neither are these images saved after taking new ones. So, since we wanted to have control over saving newly taken pictures, we went with the method described above.

### 3.3.7 Configuration

PISA needs a few configurations to be able to send (appropriate) data to the two servers in this thesis. For CEI it is very easy. There are no configurations the user must perform. All the user needs to have are the credentials to be able to log on to the CEI system, which is required to send messages to the server. The default username and password for CEI are provided in the settings of PISA, so the user can later log in with these credentials. Also, the correct IP address to send the CEI messages to is included directly in the source code of PISA, since it is static.

For sending to our wrapper server, some configuration is needed. The NFFI format, which is used for positions, requires the following attributes: `Country`, `system`, `transponder identifier (ID)`, `unit symbol` and `unit short name`. The FFI-incident format, which is used for observations, requires only a portion of the same ones as NFFI: `System`, `transponder ID` and `unit short name`. However, it operates with different names for these attributes. They are respectively called `origin system`, `ID` and `sender`. The three first attributes (country, system and transponder ID) are needed to uniquely identify the *source* of the message. The same attributes including the timestamp uniquely represent the relevant *message*.

The attribute named `country` contains of course the country where the user is located in. “NOR”, short for Norway, is a valid input here. `System` is the name of the system being used, and in our case this would be “PISA”, our mobile application. `Transponder ID` is a unique identifier of the device being used. For instance, “iPhone 4S” or “Google Nexus 7” are both valid for this type of input. `Unit symbol` is a 15 character string that contains valid APP-6A data, which is a NATO standard for map symbols. APP-6A data is only a mapping between the 15 character string and a symbol for displaying various military units. In this thesis, we only use a test value. `Unit short name` is the name of the unit out in the field, and “Squad 5” or “Platoon 7” are examples of valid data. Finally, the IP address for our server is not permanent. The server is located on the Macintosh computer used in this thesis, and the IP address changes whenever the computer is moved. However, the user has the opportunity to modify this to the correct address. Default IP address is 192.168.10.11, which is used when testing the functionality of PISA.

An example of a reported position with the NFFI format is shown in listing 3.1. The essential data lies within the `coordinates` element. The other content are standard data that is needed for the NFFI format, e.g, the `country`, `transponderId` and `dateTime`. These are mandatory elements that are required in the NFFI message. All these values are editable under the settings page of PISA. Listing 3.2 illustrates the scenario where periodic positioning is turned on, and an XML-tag containing the chosen interval is defined.

```
<track>
    <positionalData>
        <trackSource>
            <sourceSystem>
                <country>NOR</country>
                <system>PISA</system>
            </sourceSystem>
            <transponderId>iPhone 4S</transponderId>
        </trackSource>
        <dateTime>20140522173214</dateTime>
        <coordinates>
            <latitude>59.92201</latitude>
            <longitude>10.76133</longitude>
        </coordinates>
    </positionalData>
    <identificationData>
        <unitSymbol>TESTTEST--TEST-</unitSymbol>
        <unitShortName>Squad 5</unitShortName>
    </identificationData>
</track>
```

Listing 3.1: Example of Reported NFFI Position

```
<track>
    <positionalData>
        <trackSource>
            <sourceSystem>
                <country>NOR</country>
                <system>PISA</system>
            </sourceSystem>
            <transponderId>Google Nexus 7</transponderId>
        </trackSource>
        <dateTime>20140522173214</dateTime>
        <coordinates>
            <latitude>71.16546</latitude>
            <longitude>25.79917</longitude>
        </coordinates>
    </positionalData>
    <identificationData>
        <unitSymbol>TESTTEST--TEST-</unitSymbol>
        <unitShortName>Platoon 7</unitShortName>
    </identificationData>
    <periodicUpdates>
        <intervalInSeconds>10</intervalInSeconds>
    </periodicUpdates>
</track>
```

Listing 3.2: Example of Reported NFFI Position w/Periodic Updates

Listing 3.3 provides an example of how an observation containing position (latitude/longitude) and description might look like. The difference between an observation with and without a picture, can be seen by an extra `<picture>`-tag after the `<incidentDescription>`-tag at the end. These tags usually look something like the one in listing 3.4, where `[...]` in the middle of the text string is only there for visual purposes, to avoid printing a *very* long string. The content of the previous tag is an example, and just a small segment of what the tags often contain, which is a long base64 encoded text string.

```
<Incident xmlns="urn:no:ffi:reports:incident" >
```

```
    <name>Incident</name>
    <id>iPhone 4S</id>
    <dateTime>20140425120001</dateTime>
    <latitude>-27.116667</latitude>
    <longitude>-109.366667</longitude>
    <originSystem>PISA</originSystem>
    <sender>Squad 5</sender>
    <incidentDescription>I have no idea what I am doing out here.</
       incidentDescription>
</Incident>
```

Listing 3.3: Example of Reported FFI-incident Observation

```
<picture>
    data:image/jpeg;base64,AhfuYggjHJml70Q0uuNbhG3H2[...]
        I8yHng4jH0bnM2311LaaChTt007
</picture>
```

Listing 3.4: Example of a picture-tag in an Observation

The settings page of PISA, where most of the configuration is available, could contain checkboxes for choosing which of the servers to send positions and observations to. This is more desirable over multiple buttons for sending positions and observations. They could be combined into fewer buttons with the same purpose. However, due to complications with different implementation management of FFI and CEI messages, and time restrictions, this is left out from this thesis.

## 3.4 Server Design

There are two different servers PISA submits its reported data to; our wrapper server, and the CEI server. The design of these servers have some distinct differences, even though they also are quite similar in the way they communicate. Either way, they are divided into each of their own sections below, to better define their separate logic.

### 3.4.1 Wrapper Server

The server described below is a basic wrapper server implemented using PHP. The server being basic is done intentionally because of the main focus being on the application. The server's sole purpose is to work as a test server that our application can interact and communicate with. It provides only the minimum of functionality necessary to prove that interoperability with NII is easy to integrate with some further work. The data stored on the server is in XML format, and is ready for setup with SOAP and Web services to move towards NII. Therefore, requirement number ten in table 2.1 is fulfilled by design. This server can be used as a gateway between other Web services back-end infrastructures, since it facilitates for interoperability towards for instance NII on the communication level.

The communication pattern between PISA and our server is illustrated in figure 3.8. It shows the basic message exchanges between our app, on either iOS or Android, and our server. PISA is a platform independent application that must push/deploy the latest functionality first, to an arbitrary device on any of the two platforms, to work. This is done prior to the communication with the server, at deployment time. The communication between PISA and server occurs at run-time. We have two different messages we send to the server, and those are NFFI (positions) and FFI-incident (observations). Both of these messages are sent using the REST interface. Each time the application reports either a position or an observation, the server replies with an appropriate response. This is to let the user know if the data was successfully received at the server, or not.

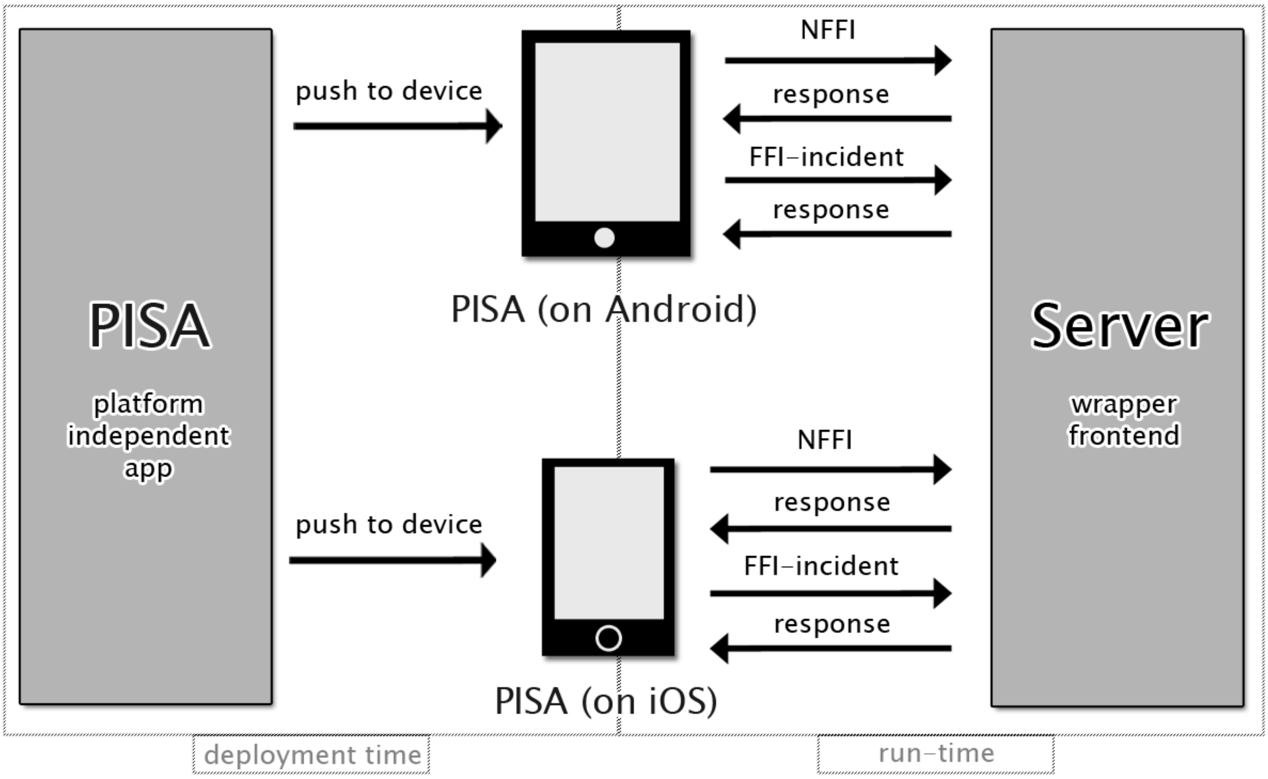


Figure 3.8: Design of Server Communication Pattern

NFFI is the format used to report positions to our server, both on demand and periodic ones. Whenever a position is successfully received, the server responds with the text

"Your position was received at the wrapper server". The position is stored in an XML file on the server. This XML file has a predefined format, so the XML is constantly be valid. The two different types of positions are both stored on the same file. If the file exists from before, all the new positions are appended at the current end of the file. If it does not exist, a new file is created.

The other format used, is the FFI-incident format. This is used for reporting observations (possibly containing a position), but not positions alone. The NFFI format is already used for this functionality. Observations contain a description of the observation, and optionally also a position and a picture. The content of the message varies depending on the data being sent. This is taken care of on the application side. When an observation is successfully received, the server responds with the text "Your observation was received at the wrapper server". The observations are stored in an XML file on the server in a predefined format different from the NFFI messages. All three different combinations of an observation are stored on the same file. If the file exists, new observations are appended at the end of the file. If not, a new file for the FFI-incident observations is created.

### 3.4.2 CEI Server

The previous section describes how the communication goes between PISA and our wrapper server. However, we also have the second scenario where we want PISA to communicate with our external server, the one provided by CEI. The communication pattern between these entities can be seen in figure 3.9. We recognize its similarities with the previous figure, but the messages and its content are quite different. We also have one extra message compared to figure 3.8.

The CEI server is more complex and complete in relation to our wrapper server. It includes more functionality, and provides several interfaces. We use some of them, and the rest are out of this thesis' scope. The server accepts data built in JSON formats, in addition to other data structures. However, XML is not one of them.

Figure 3.9 is very similar to figure 3.8 in the previous section. PISA, the wrapper application, and the devices are the same. The server is now CEI's own server, and not ours. The messages in this new figure are different, so these need to be explained. The response messages are standard HTTP responses. They let the user know if a position or observation is successfully created, or if there were problems with the authorization, server or current request. In appendix B, table B.1 illustrates the error messages sent from the server, and table B.2 shows the success messages sent from it.

In this case, we also have two main data groups we send to the server. Those two are *positions* and *observations*. Both of these messages are, like before, sent using the REST

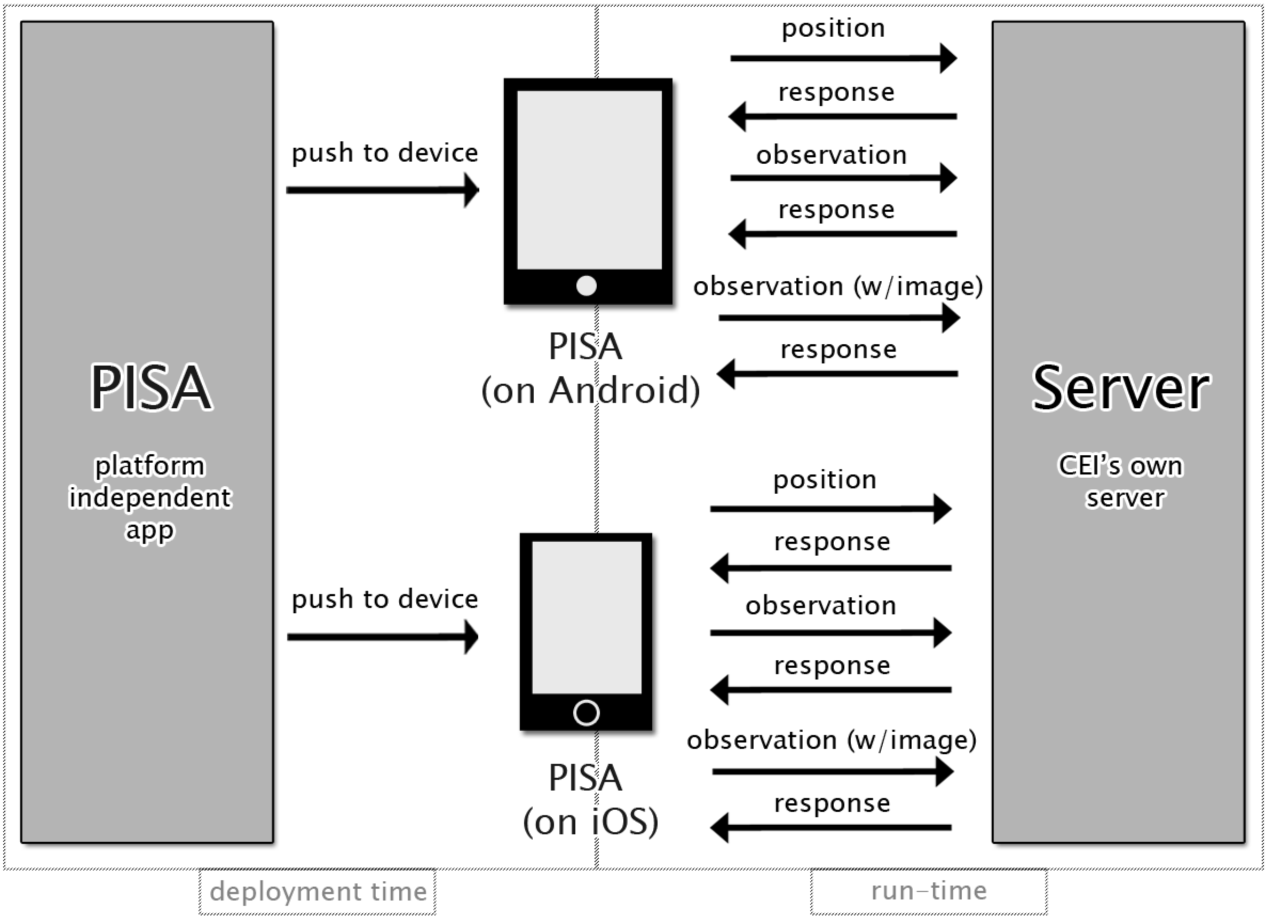


Figure 3.9: Design of Server Communication Pattern (with CEI)

interface. JSON is the required format of the registered data. However, a third message is necessary to be able to send observations containing an image file. We therefore have three different potential messages we send to the CEI server.

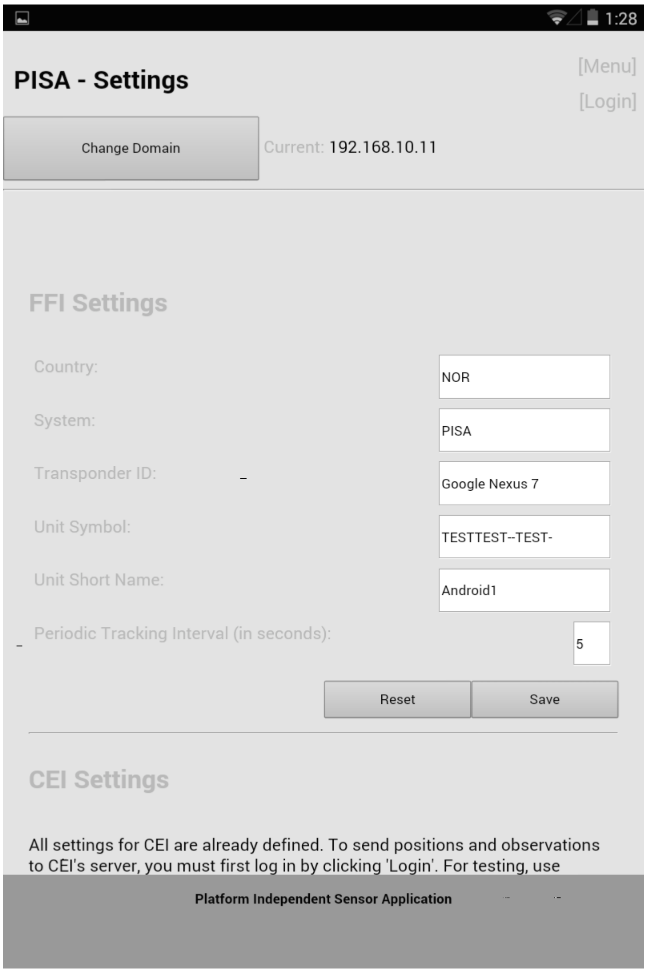
*Positions* are, as the name implies, positions/locations reported to the CEI server. The positions are stored on the server. The positions are sent as JSON data objects. The other format used is the *observation* format. This is used for reporting observations, containing position and description. These observations are built and sent with the JSON data format, and they are stored on this server. The last message, *observations with image*, is not registered as a JSON object, but posted as a web form submission. These messages are of course also saved on the server. All of the messages received at the CEI server, positions and observations, can be viewed in CEI's web application.

## 3.5 GUI Design

In this section the GUI, and generally the user interface, of PISA is described and discussed. We explain how the app looks and justify why it looks like this. Other alternative ways of the design are mentioned, and an explanation to why these were not chosen is provided.

Designing a good user interface is challenging, and is its own area of research. As this was not the primary focus of the work of PISA we have not made an exhaustive study, but we have rather chosen an approach where we base our design on experiences from other projects, primarily CEI, which has developed applications for use by military units at the tactical level. To get an overview of the experiences that have been made during the work with CEI, we based ourselves on both CEI's documentation [10] and discussions with one of CEI's developers [39]. We use these experiences as guidelines for our own GUI, and try to follow these recommendations from previous work as far as possible. Below we present these guidelines, and what we have done to ensure that our GUI design follows them.

The first design rule is to beware of having too small font size on most of the text used in the app. Since PISA is designed to be used on different devices types, ranging from compact smart phones to larger tablets, the GUI must function equally well across a range of different screen sizes and resolutions. It is important that the user does not spend all focus on squinting on the mobile screen to read the small text. It is also essential to get an overview of the application quickly, due to pressure or stress if the user runs into an incident of importance. We have used an appropriate font size of all the essential text in PISA. Figure 3.10 provides a screenshot of PISA where we see some of the text displayed to the user.



PISA - Settings

Current: 192.168.10.11

FFI Settings

Country: NOR

System: PISA

Transponder ID: Google Nexus 7

Unit Symbol: TESTTEST--TEST-

Unit Short Name: Android1

Periodic Tracking Interval (in seconds): 5

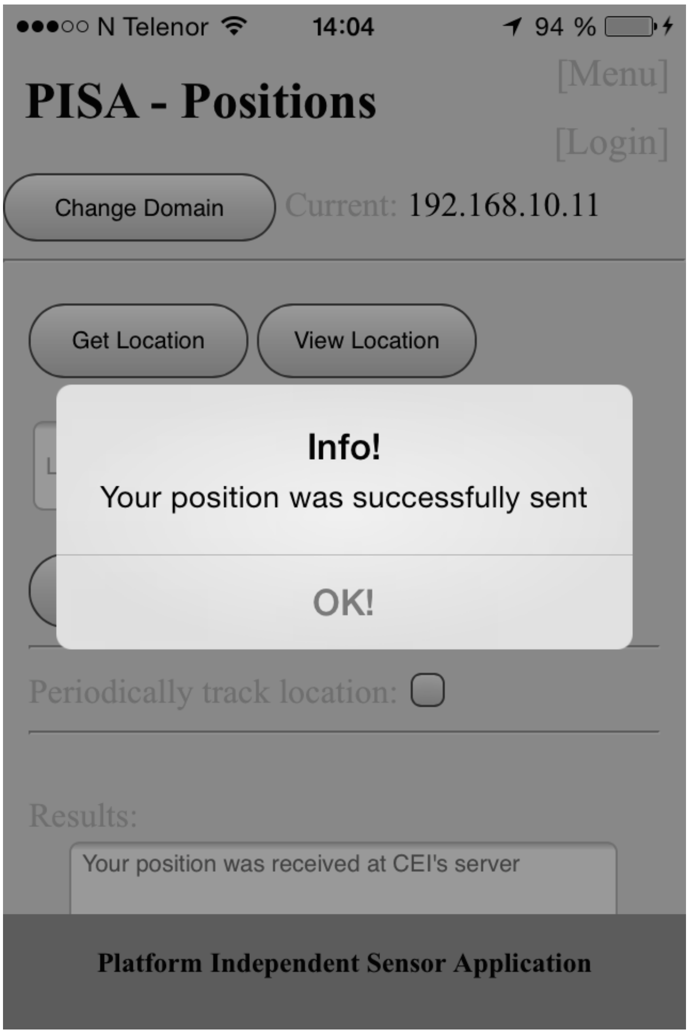
CEI Settings

All settings for CEI are already defined. To send positions and observations to CEI's server, you must first log in by clicking 'Login'. For testing, use

Platform Independent Sensor Application

Figure 3.10: Example of Text Sizes in PISA

Another design guideline is to be careful not to display too much text or information to the user. Since the user often could be in a stressed situation, and if the information is not short and concise enough, this may result in the user failing to register vital details. With PISA, we operate with as concise messages as possible, letting the user understand the essential information with as little text as possible. An example of some text intended for the user is shown in figure 3.11.



Current: 192.168.10.11

Info!

Your position was successfully sent

Results:

Your position was received at CEI's server

Figure 3.11: Example of Text Amount in PISA

“Fat fingers” could be a problem when operating applications on handheld devices. Users can be out in the terrain walking or running, or they could be wearing gloves. If equipment with touch screens become standard gear, this of course requires that the potentially used gloves are designed for use with touch screens. This leads to a problem if the buttons they click on are too small or too close to each other. Therefore, big buttons are necessary in order to overcome the "fat fingers" issue. We generally try to have large buttons in our app, without making it feel abnormal. The buttons are enlarged to some degree, and we have different sizes on different groups of buttons, depending on how vital they are and how much they are believed to be pushed. Figure 3.12 refers to PISA's menu, where we use big buttons.

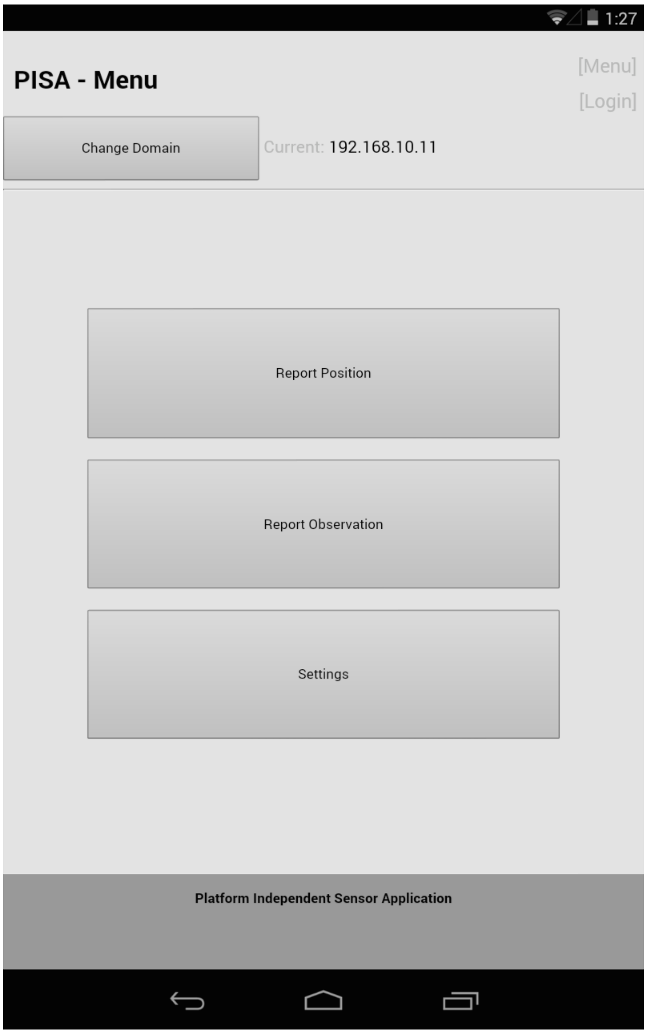
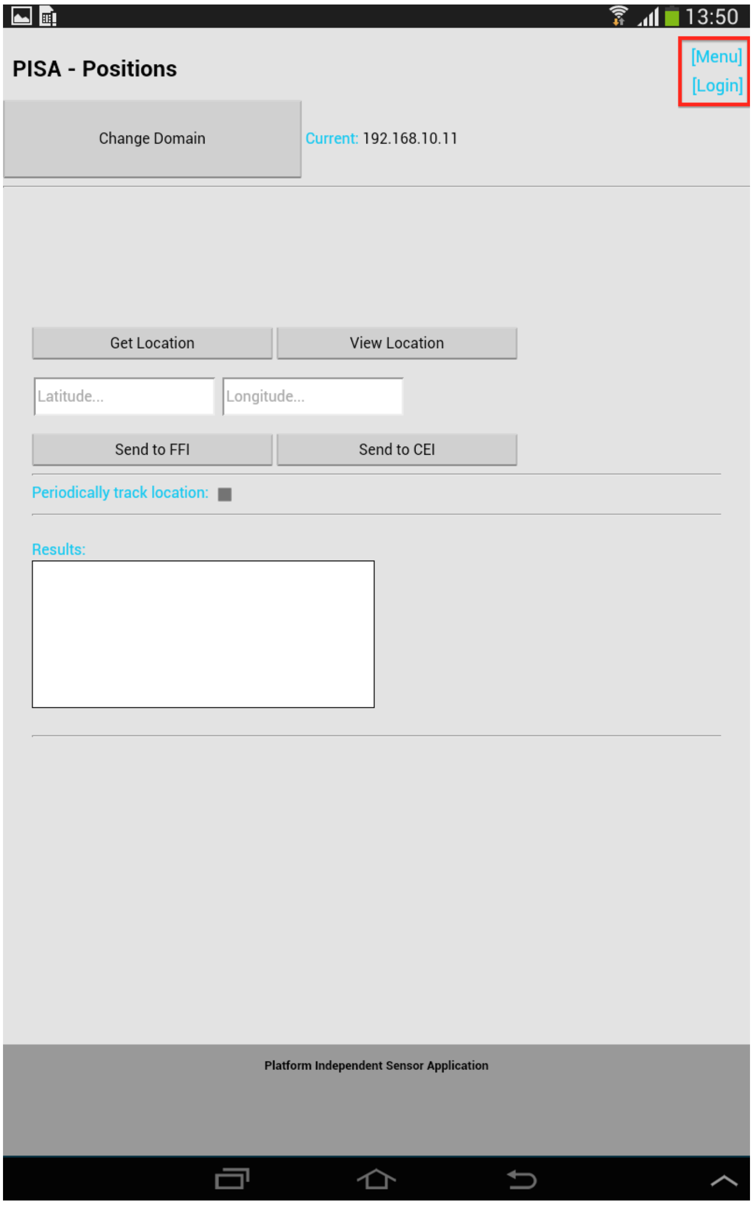


Figure 3.12: Example of Big Buttons in PISA

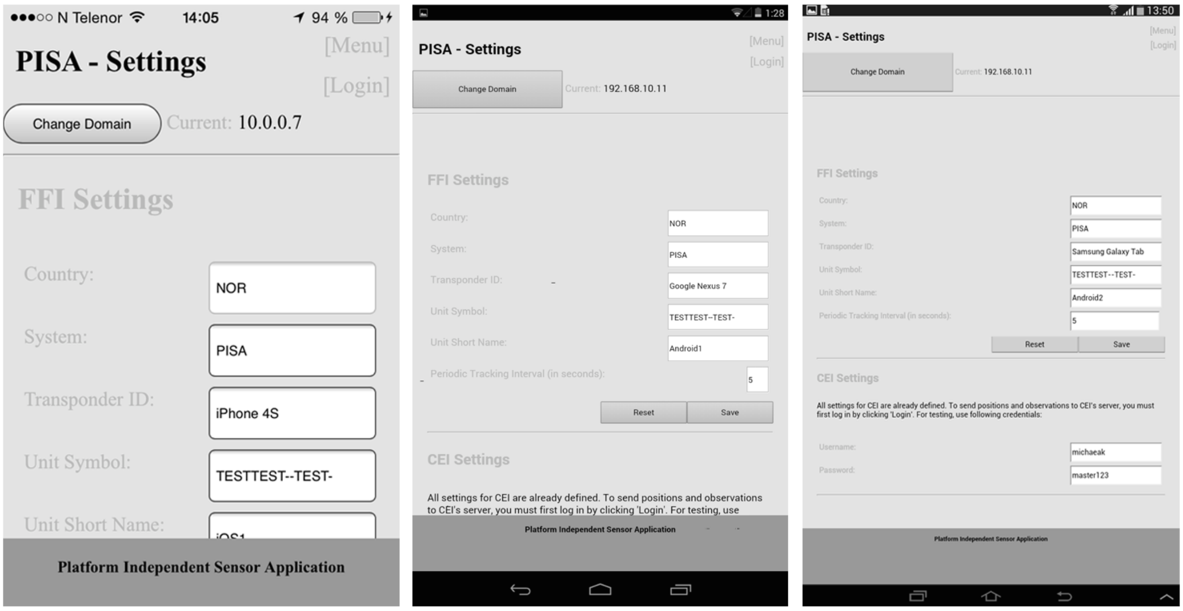
Easy navigation is also a requirement for a good user experience. Easy navigation between pages is required to make the user feel that he always can get to the essential pages with just a few clicks of a button. The navigation must be clear and visible, and also easy to press so the user does not have to feel any additional pressure if already in a stressed situation. A navigation button called “Menu” is included in our application. This always takes the user back to the starting position of PISA, which is the menu. Since PISA does not consist of a large amount of different pages, but only a few, this button might be all that is necessary to navigate the user in the most optimal way. This button, along with some others, can be seen from figure 3.13.



Current: 192.168.10.11

Figure 3.13: Example of Navigation in PISA

Responsive design is also an important aspect to consider when implementing mobile applications. Screen sizes and screen resolutions vary with every handheld device, especially when it comes to the difference between smartphones and tablets. Smartphones are of course usually smaller, and tablets have bigger screen sizes and possibly greater screen resolutions or different dimensions. This application focuses on being deployed to iOS and Android devices, where the range in screen sizes could be quite big. Our design mostly includes percentage values when it comes to the size of the different graphical components of the app, thus resizing the text, buttons and input fields according the each screen size or resolution on its own. However, due to time restrictions and that responsive design did not get that much focus in this thesis, the application looks different on each device. With the time we had, we wanted to make the app feel normal on every screen size. An example of how PISA looks on both the iOS and Android devices is provided with figure 3.14. In the illustration, all screenshots are resized to be equal in size. The figure therefore illustrates the importance of adjusting the amount of information displayed to fit the screen size. The leftmost figure looks relatively good, whereas the others contain too much information to function well at this current resolution. In real life, however, the respective devices are bigger, thus allowing this to actually work fairly well.



Current: 10.0.0.7
Country: NOR
System: PISA
Transponder ID: iPhone 4S
Unit Symbol: TESTTEST--TEST-
Unit Short Name: iOS1

Current: 192.168.10.11
Country: NOR
System: PISA
Transponder ID: Google Nexus 7
Unit Symbol: TESTTEST--TEST-
Unit Short Name: Android1
Periodic Tracking Interval (in seconds): 5

All settings for CEI are already defined. To send positions and observations to CEI's server, you must first log in by clicking 'Login'. For testing, use

Current: 192.168.10.11
Country: NOR
System: PISA
Transponder ID: Samsung Galaxy Tab
Unit Symbol: TESTTEST--TEST-
Unit Short Name: Android2
Periodic Tracking Interval (in seconds): 5

All settings for CEI are already defined. To send positions and observations to CEI's server, you must first log in by clicking 'Login'. For testing, use following credentials:

Username: michaeak
Password: master123

Figure 3.14: Design of PISA on Different Devices

The main functionality of the application should be easily available, and it should be simple for the user to understand what the app basically does. This is essential for helping the user understand the area of use this app is meant for. This is why we wanted to highlight and emphasize the position and observation aspect of our application. This is the reason for why we have “Report Position” and “Report Observation” as two of the three big main buttons on our menu, which can be seen in figure 3.15. The main functionality should also be fast and easy to use, while secondary functionality like the configurations is more accepted of being slower and more advanced. This is because users might adjust the settings on the app before they head out on a mission, while the main functionality could be repeatedly used when in a stressful situation. Therefore it is quite important that this functionality is fast, and easy to locate and use.

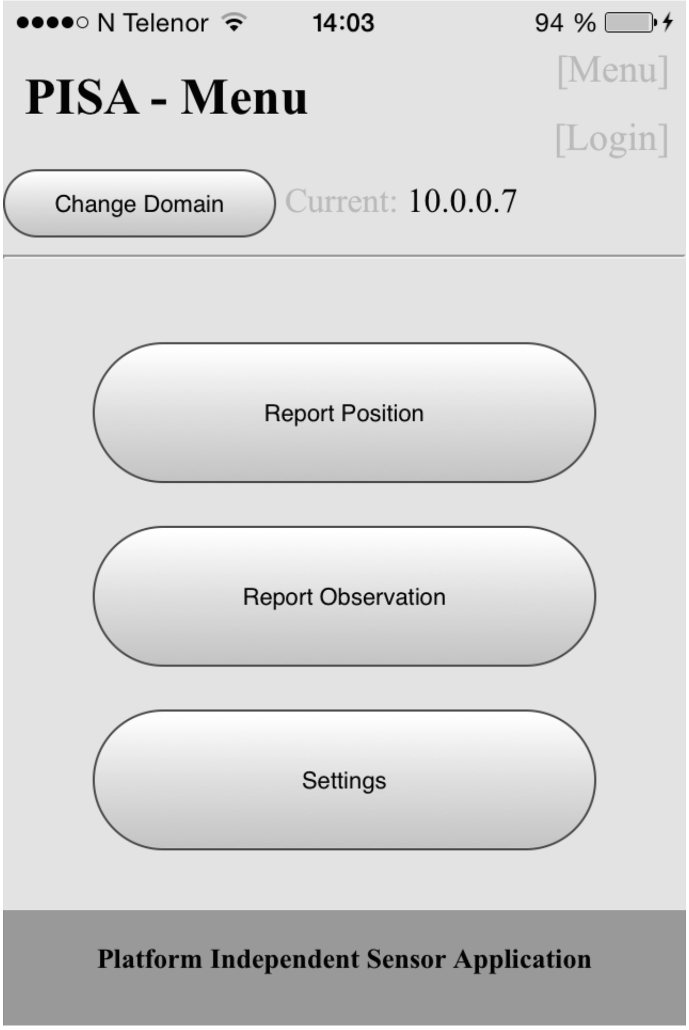


Figure 3.15: Visibility of Main Functionality in PISA

Novices and more experienced users have different requirements when it comes to how the functionality of an application is presented. For novices to quickly get started with the use of an application, it is important that the app is intuitive and self-explanatory, so that users understand what needs to be done fast. For a more experienced user, it is often more important that the application is efficient in use, meaning that the functionality that is often used is quick. Examples of such effective use is to have few clicks and little scrolling to carry out a task. It is not always possible to take full account of both these factors simultaneously, and in PISA we have chosen to focus mainly on novice users and having an intuitive design. Professional (enlisted) soldiers, i.e., the army, will typically have received a lot of training. The Home Guard who might be most appropriate for this type of app will *not* have the same resources for training. They practice indoors for about a week of the year. Regardless, it is better for all soldiers to spend less time learning communications equipment, and more time practicing professional military aspects.

Another important thing to remember, is that the application should work equally well on both iOS and Android, and to some extent it should also look and feel the same. What we must keep in mind is that some users are very comfortable with one of the mobile platforms, and maybe not at all with the other. The iOS and Android operating systems are quite different from each other, and native apps on each platform could feel very different. So it is important to try and make a kind of neutral application that both iOS and Android users feel comfortable using.

## 3.6 Summary

This chapter presented the design and implementation details of both our application, PISA, and the servers we communicate with. The design was conducted according to the requirements specification in chapter 2. Both an overall design and a detailed design over our whole solution was provided, describing how the different components communicate with each other. PISA communicates both with our wrapper server, and the server provided by CEI. So instead of only communicating with our own implemented server, PISA interacts with a server belonging to another existing prototype (CEI).

In addition, both the design and implementation of our application functionality were presented. Here we looked at the different aspects of the functionality of PISA, and explained how they were designed and how we implemented them. The on demand positions, periodic positions, and all types of observations were discussed in detail. Some general functionality was also presented, along with the configuration part of PISA.

The design of the servers were also provided. The formats they use, and how they communicate with our application were discussed. Our server, which we built as a wrapper towards NII, was included, along with CEI's own server. After the two servers were described, the design of the GUI was presented. When we designed the GUI design, we had several design guidelines we tried to follow. These guidelines were based on our own and CEI's experiences. Screenshots of our application were added to illustrate what we actually implemented.

Requirement number seven and ten from table 2.1 were fulfilled by design in this chapter. We have set the prototype solution to always use REST as the communication method, and so therefore requirement number seven was satisfied. Requirement number nine was fulfilled by using the NFFI and XML formats, even though we also use proprietary formats like the FFI-incident and CEI's proprietary JSON formats. FFI-incident was used since NATO does not have an unambiguous standard for sending observations, and CEI dictates their own formats. This requirement was not compromised by this, since it stated that we used standards where *possible*. This we did, it was simply not possible everywhere. Nor was this requirement that essential, due to having a priority level of three. Requirement number ten was fulfilled by utilizing the NFFI and XML formats, including facilitating for the wrapper server to be used as a gateway towards NII or SOAP Web services infrastructures.

Evaluation and testing details are provided in the next chapter.

This page is intentionally left blank.

# Chapter 4

# Evaluation

In this chapter, we evaluate PISA and discuss the results. The evaluation was performed in three phases: Function tests, DIL tests and GUI tests. For evaluating the results of each test, we use the following terminology: Success, limited success and failure. Success is used for when the test is completely successful and works every time. Limited success means that the test is successful sometimes, or at least some portion of the test is successful. Failure is of course when the test is unsuccessful and fails every time. First, however, the evaluation tools used in this thesis are presented, before the phases, including the individual tests each phase consists of, are discussed.

## 4.1 Evaluation Tools

Both real devices and emulators/simulators were used for evaluating our prototype application. It should be noted that while the two terms emulator and simulator are commonly used to describe two different approaches to testing [43], they are often used interchangeably when discussing mobile application development. In this context they are both used to describe a piece of software that mimics the run-time environment of a given mobile device, and can be used for testing applications without having to deploy them on a physical device. Thus, the most correct term to apply to this type of software would be emulator. However, the iOS emulator is called "iOS Simulator", and the Android emulator is more properly named "Android Emulator". Because of this naming confusion, when the "iOS Simulator" is mentioned in the thesis, we in fact mean emulator.

For developing purposes, emulators were mostly used. This is due to the fact that emulators are a simple and fast way to test an application's functionality, e.g., when lacking real devices to test on. Emulators are often used as a preliminary testing tool to use before testing the application on an actual device. Some important details about the emulators used in this thesis are provided in the following sections below.

### 4.1.1 The iOS Simulator

The iOS Simulator (which in reality is an emulator) is provided when installing the Xcode tool on a Mac. Xcode, including this emulator, is completely free of charge and is downloadable from the App Store [18]. A Software Development Kit (SDK), which gives all the necessary tools for building an iOS application, is provided when downloading the newest version of Xcode.

The emulator was used throughout the development of the application. We used a real iOS device when we get to the testing of our solution, because it is important to see that the device behaves as expected. The emulator often has some limitations when it comes to features and hardware. However, the iOS Simulator is quite fast and responsive, in addition to support most of the major functionality on a real device. It can also mimic several gestures like pressing the home button, shaking the phone, switching orientation, etc.

### 4.1.2 The Android Emulator

The Android Emulator is provided when installing the Android Developer Tools (ADT) from the Android Developer site [21]. This download contains the Android SDK, an Android version of Eclipse and of course the emulator, among others. When downloaded and installed, you may choose whichever physical device and OS version (API level) you want to emulate a virtual clone of. This can be done from a component called the Android SDK Manager, and further the Android Virtual Device (AVD) manager.

The Android emulators have a bad reputation of sometimes being painfully slow, particularly if you have a computer that does not have the best specifications. The emulators support a lot of the features and gestures available on the corresponding Android devices, but due to the very long boot time and slow execution time, testing on a real device is likely to be more time efficient. Because of this, real devices were used more than the emulator during this thesis.

### 4.1.3 Real Devices

Emulators were used much during the implementation of the app, but to better evaluate how the application works in relation to the thesis' set requirements, we want to test PISA using real devices. The devices used for testing in this thesis are shown in table 4.1.

The table above may include some unfamiliar terminology, and warrants further explanation: The far right column tells if the appropriate device has a corresponding emulator. The “Display”-column provides the device's screen size in inches, then the screen resolu-

| **OS** | **Manufacturer** | **Model** | **OS version** | **API level** | **Display** | **Emulator** |
|---|---|---|---|---|---|---|
| iOS | Apple smartphone | iPhone 4S, 16GB | 7.1.2 | - | 3.5" (960x640) | ✓ |
| Android | Google tablet | Asus Nexus 7, 32GB | 4.4.3 (KitKat) | 19 | 7" (1280x800) | ✓ |
| Android | Samsung tablet | Galaxy Tab 2 10.1, 16GB | 4.2.2 (Jelly Bean) | 17 | 10.1" (1280x800) | ✓ |

Table 4.1: Testing devices

tion in pixels (in parenthesis). The API level is terminology reserved for Android devices only, since iOS does not operate with this. API levels are intended for developers, and it is an integer value which represents what built-in functions and functionality are available for the developer. The higher Android OS version, the higher API level. As the level increases, offered functionality adds up. The rest of the columns (starting at the left) describe the devices' OS, manufacturer and type of device, model and OS version. Only Android have named their OS versions.

We have chosen to test two different devices using the Android platform. This is because we want to test various manufacturers, different screen sizes and to see if there are any variations of the Android OS's. We do not test different devices of iOS, since there is only one manufacturer behind that OS: Apple. They make both the hardware and the OS for all iOS devices. The amount of different versions in use is therefore lower than for Android. Since Android is used by various manufacturers (that often have their own GUI, support functions, etc. on their devices), combined with the fact that not all devices support the same Android version, means that there are much bigger variations in the OS version (thereby API, functionality, etc.) on Android.

To use an iOS device and deploy apps on it, you are required to have an iOS Developer License. This costs $99 a year, and for novices it can seem difficult and a large amount of work to configure and install all necessary components.

With Android, it is not much extra work to deploy your apps to a device. You only need to set your device to “developer mode” and have a mini-USB cable. It is not much work to put your Android device to “developer mode”. This is easily done directly in the device's settings.

## 4.2 Function Tests

The first phase of testing performed was the function tests. In this phase we tested PISA without any forced errors or limitations like connectivity issues, packet loss, limited bandwidth, etc. What we mean by this is that no emulated or manual errors are intentionally introduced to the network. With functionality we focus on the transmission of positions and observations to our two servers.

### 4.2.1 Objective

The objective of this test is to ensure that the app works *before* we start testing it in a DIL environment. If everything works in the function tests, and it fails in one of the DIL tests, then we can say with certainty that the error has a connection with the introduced network, and not with the application itself.

### 4.2.2 Execution

We execute these tests to both our own server and the CEI server. We test the application over WiFi in a regular Local Area Network (LAN) with normal attributes and network traffic. Both Android and iOS devices are used for testing. Figure 4.1 shows the test scenario, where the two cases represent each of the servers being transmitted to.

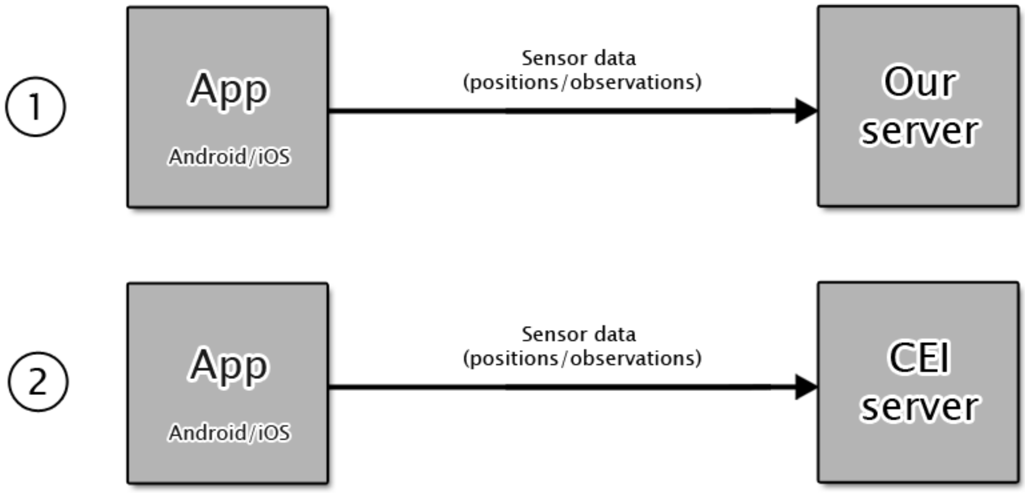


Figure 4.1: Function Tests

The functionalities being tested in this test are the ones described in the design chapter, more specifically under section 3.3. We test the sending of on demand positions, explained in section 3.3.2. The transmission of periodic positions is also tested, however only to our wrapper server. This functionality is defined in section 3.3.3. In addition, we of course test the sending of observations. Observations can be sent with description (only applicable to our server), description and position, and the previous two including an image. The functionality of these types of observations are described in section 3.3.4, 3.3.5 and 3.3.6 respectively.

We expect that all the messages sent by all devices are successfully received by both servers in this case. This is because we do not expect any major network errors, since this is a functionality test. The limitations should not lie in the network, because we want to test that the application works before we test it in different limited networks.

The functionality mentioned above is the functionality being tested for *every* following scenario in this chapter. In all cases, we test the on demand positions, periodic positions and all three types of observations.

### 4.2.3 Results and Analysis

During the execution of these tests, we observed that all of the message transmissions were successful. Both types of position updates (on demand and periodic) and all types of observations were sent from PISA, and were successfully received by both servers. All three devices had the same exact behavior, and nothing was out of the ordinary. Transmitting on demand positions and observations with either description only or description and position went smoothly in a matter of seconds or less. The registration of periodic positions also went well, where all positions were successfully received at the servers. The observations including an image took about the same amount of time to send, due to no restrictions on the network. We also verified that PISA provided correct feedback to the user, so that he always was informed about what was happening. The user received pop-ups that confirmed that the messages were sent, that periodic positioning was on or off, and which interval they were sent, etc. The results are summarized in table 4.2.

The observations we made show that this test is definitely successful. These function tests partially confirm requirements number five and six from table 2.1, with respect to the fact that the application itself handles both message types with the expected behavior. In order to fully confirm that these requirements are met however, we need to test the application in an environment where loss is present.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Success | Success |

Table 4.2: Results of the Function Tests

## 4.3 DIL Tests - Disconnected

Phase two of the testing of PISA involves testing the application in a DIL environment. Here, we want to verify that PISA works in a DIL environment, and we have chosen to divide these tests into three separate stages: One for the disconnected part (D), one for the intermittent part (I) and finally one for the limited part (L).

In this first test we are evaluating the disconnected part of DIL environments. And with disconnected, we mean that the network suddenly goes down when we try to send data to the server, and connection loss is introduced to our scenario.

### 4.3.1 Objective

The goal here is to verify that PISA handles being disconnected, and that essential data is properly kept. When the user tries to send a message, and the connection is lost before the message is successfully received at the server, we want PISA to still be working. The app is required to work without a network connection, and that is why we test this. During a disconnection, PISA should not ruin the user experience by freezing, or in some other way prevent the user doing other things. It should not crash either, and PISA should maintain control of how many of the periodic positions are lost. When on demand positions and observations are not successfully sent, the user should be informed of this and the data should be cached for later retransmission. The unsuccessful periodic positions are only counted as losses and presented to the user, i.e. they are not stored for retransmission later.

There are two ways of the connection being lost in this scenario: The connection could go down on either the server side or on the application side. Regardless, both events are tested below. It is important to test this due to them being two different states from the device's point of view. The first one is loss of network connection, the other one is that the server does not respond. These are two different states on the device, and we need to verify that PISA performs as expected independent of the type of network connectivity issue the device is experiencing.

### 4.3.2 Execution

Once again, we test the application in a LAN with normal attributes and traffic, but we introduce a connectivity loss in two ways. In the first case we disconnect the server from the network, making the host unreachable. In the second case we simulate a connectivity loss on the device itself by turning off the WiFi and/or mobile data, or setting the device into flight mode.

All three devices are tested against our own server. An illustration of the test case can be seen in figure 4.2. We only test against our server, since CEI only is available over the Internet. Because of that, we do not have any control over the CEI server. This further leads to unpredictable network behavior, so the testing could have been inconclusive. Therefore, we have to test this scenario against our wrapper server. The subsequent scenarios in the following sections are also *only* tested against our server due to the reasons described above.

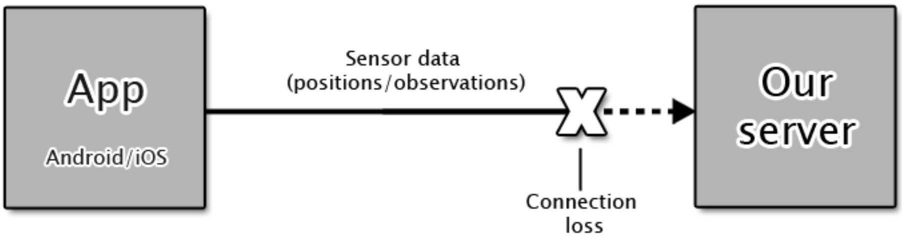


Figure 4.2: DIL Tests - Disconnected

It is the main functionality that is tested in this scenario. We expect that every single one of the messages, except the periodic positions, are unable to be sent in this scenario due to the network connection being lost. All these messages should be cached by PISA when the network connection is lost. The user should still be able to do other things, but the latest message is stored on the device until the user may be able to send the message at a later point. The user is also be presented with an error message, letting him know that the message failed to be sent. The periodic positions, however, are lost. This is because they are the only loss tolerant messages, i.e., loss is accepted with periodic positions. The lost periodic positions, due to connection loss, are only counted and presented to the user as they are lost.

### 4.3.3 Results and Analysis

We observed that all of the message transmissions that failed to be sent due to connection errors, were either stored on the device or counted as failed positions (with periodic positions). The sending of on demand positions and all types of observations, were successfully taken care of by PISA. All data that did not tolerate loss, was always kept by the application and a message telling the user of this information was displayed. The loss tolerant data, i.e. the periodic positions, were not cached, but instead counted as failed positions and each presented to the user as they fail when a network error occurred.

There were also no big differences in the results between the devices. Of the messages that did not tolerate loss, none were lost. The other loss tolerant messages that were lost were counted and presented to the user. The results from this test are summarized in table 4.3.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Success | Success |

Table 4.3: Results of the Disconnected DIL Tests

The observations show that the test is successful. These DIL tests therefore verify requirement number four in table 2.1, when it comes to the disconnected aspect. Requirement five is also verified because loss tolerant data is handled when the connection goes down. Requirement six is partially verified due to the essential data being cached on the device. Requirement number six is only partially verified since the data is stored, but we do not know if they are successfully transmitted until the connection is reestablished. We look at this scenario in the next section. Since all messages are either cached or accounted for when the connection goes down, this means that the disconnected DIL tests are successful. It is also successful since the application never froze or crashed, the user was free to do other things while the network was down.

## 4.4 DIL Tests - Intermittent

In this scenario we lose our connection to the server, and then regain it again, when we try to transmit data over the network. Here, the user discovers that the network is up again, and this enables him to retransmit the data which has been stored on the device. The Intermittent (I) part of DIL simply means that our connection to the server may be lost, but then reestablished again at some later point in time. This scenario builds upon the previous one.

### 4.4.1 Objective

The objective here is to verify that PISA not only keeps the data that it was not able to send when the connection went down, but also enables the user to resend this data at will when the connection is reestablished.

The user must discover that the network is back on PISA by manually checking if the device has a network connection again, and then try to resend the message. This is a design choice, and it is explained in section 3.3.1. The user can also try to send previously stored messages when the connection was lost at the server. However, the user has no way of knowing if the server side connection is reestablished. The user just has to try to send the message again. If the network is properly up and running, the message is finally sent. A message telling the user that the message was now successfully sent is then displayed.

### 4.4.2 Execution

Once more, we test the application in a normal network (LAN) where we here introduce the intermittent aspect by taking down the connection on both sides, in addition to later bringing the connection back up again. We also only test against our wrapper server. Both types of our mobile platforms (and all three devices) were utilized for testing. Figure 4.3 illustrates the test scenario.

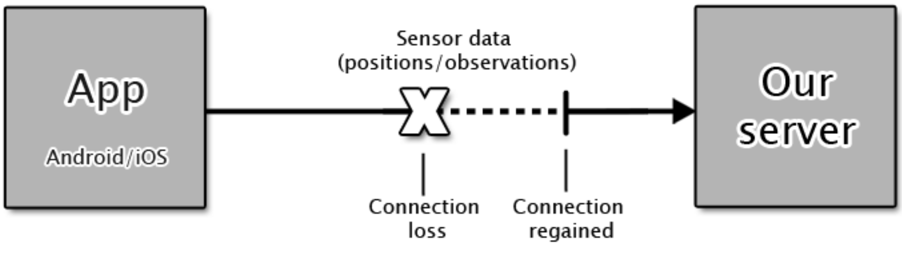


Figure 4.3: DIL Tests - Intermittent

Here, the main functionality is also tested. Since this scenario is the natural pick up point from the last one, we operate with two different ways of losing connection here also: On the server side, and the application side. Both events are tested.

We expect that every single one of the messages, except from the periodic positions, have been stored on the application since the connection was lost. The user has been able to do other things, but can now send the previous message since a network connection is available again. The user is also presented with a message, telling him that the message has now been successfully received at the server. The periodic positions, due to the reestablishment of network connection, are still counted and presented to the user once they are updated by the specified interval in the settings of PISA. However, now it resumes counting successful periodic positions again, and not failed positions. For example, if five seconds is specified, the counted successful/failed messages are updated every fifth second and always displayed to the user. This way, he always knows how many positions have failed, and how many have been received at the server.

### 4.4.3 Results and Analysis

We observed that all of the message transmissions that failed to be sent due to connection errors, were either stored on the device or counted as failed positions (with periodic positions). When the connection was back, the user could successfully send the cached messages, and the periodic positions were counted as successful positions once again. However, the messages are not considered successful until they are *received* at the server. A message letting the user know the previous message that was stored due to connection errors, was now sent and received at server, would be displayed. A message of the periodic positions sent and lost would be presented to the user.

There were no variations in the results between the three devices. The messages that did not tolerate loss, were retransmitted and received at the server when the connection was back up after an arbitrary downtime. The loss tolerant messages on the other hand, were counted as failed messages during downtime and as successful messages when the connection was reestablished. Results are summarized in table 4.4.

The observations show that the test is successful, and our expectations were correct. The DIL tests performed in this scenario verify requirement number four in table 2.1, now regarding the intermittent aspect of DIL. Requirement five and six are now also verified due to the appropriate data being handled correctly. Since all messages are now resent or taken care of when the connection goes down and back up again, this means that the intermittent DIL tests are successful.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Success | Success |

Table 4.4: Results of the Intermittent DIL Tests

## 4.5 DIL Tests - Limited

The Limited part of DIL is more complicated to illustrate than the two previous parts. This is because the network can be limited in several ways. High delay, high packet loss and low data rate all represent "limited" aspects of a network. We need to test and evaluate how the application behaves when introduced to networks with limited properties. We must test if positions and the different types of the observation messages behave as desired. When packets are lost, the data rate is low and/or the delay is high, the various messages may fail to reach their destination. This is what we evaluate in the subsequent sections, i.e., if our application handles the different capabilities of the limited DIL-networks.

We have chosen to evaluate these aspects by emulating them with a network emulating tool. We have used a FFI tool that is based on the Linux utility called "netem". The tool provides functionality for setting various kinds of restrictions on the network, and we make use of the opportunity of adjusting three different parameters: Data rate, delay and Packet Error Rate (PER). Note that these are not the only ways in which a network can be limited, but they represent common network restriction types in mobile networks. In addition, these parameters are the ones identified by NATO STO/IST-118 "SOA Recommendations for Disadvantaged Grids in the Tactical Domain". For more details on these characteristics, see table 4.5.

We have chosen to test PISA over five different emulated network configurations. These are listed as the following sections below, which includes results from transmitting the different data over the various networks. The essential characteristics of each network are summarized in table 4.6. More information about the networks are provided in each of their respective sections. It should be noted that there are almost an infinite number of possible network combinations. We have chosen to utilize the same ones as those identified for DIL-testing in NATO STO/IST-118.

| **Characteristic** | **Description** |
|---|---|
| Data rate | Data rate is the amount of data that can be transmitted over a network per unit of time. It is often used to define how “fast” a network is. Another name also used for the same concept is the network’s “bandwidth”. Data rate is quantified using bits per second (bit/s or bps), often in conjunction with an SI prefix like kilo (kbps), mega (Mbps) and giga (Gbps). |
| Delay | The delay of a network specifies how long it takes for a bit of data to travel across the network from node to node. It is typically measured in fractions of seconds. Milliseconds (ms) or nanoseconds (ns) are very much used. Transmission delay or packet delay, which we use, is the time it takes to transfer a packet from the sending node to the receiving node. |
| Packet Error Rate (PER) | PER is the number of incorrectly received data packets divided by the total number of received packets. A packet is declared incorrect if at least one bit is erroneous. Packet errors could mean corrupted, lost or duplicated packets. PER is often measured in percent (%). |

Table 4.5: Explanations of Network Characteristics

| **Network** | **Data Rate** | **Delay** | **PER** | **Type of Network** |
|---|---|---|---|---|
| Satellite Communications (SATCOM) | 250 kbps | 550 ms | 0 | Transit |
| Line of Sight (LOS) | 2 Mbps | 5 ms | 0 | Transit |
| Wireless Fidelity (WiFi) 1 | 2 Mbps | 100 ms | 1 % | Tactical edge |
| WiFi 2 | 2 Mbps | 100 ms | 20 % | Tactical edge |
| Combat Net Radio (CNR) with Forward Error Correction (FEC) | 9.6 kbps | 100 ms | 1 % | Transit |

Table 4.6: Overview of the Different Network Characteristics

A route through a network between a device and the server it communicates with, will in many cases consist of multiple networks that must be traversed to be able to communicate properly. Generally, we can distinguish between something called edge networks and transit networks. Edge networks are the networks the end user is connected to. Transit networks are networks that the information must cross to reach its destination.

In the scenarios we look at (tactical military), the bottleneck can be either in the edge network or in the transit network. Where the bottleneck lies is dependent on the network configuration in each case. We wish to evaluate different scenarios, independent of the bottleneck in the communication being in the edge or the transit network. Because of this, we have chosen an approach where we identify the network types that are potential bottlenecks in the communication from tactical edge and into the central network infrastructure. By testing our application over each of these network types, we can see the effect of every type of bottleneck, independent of whether the network is a typical edge network (WiFi), a typical transit network (LOS and SATCOM) or a mix of both (CNR). We run the application over emulated network configurations which represent each of these bottleneck networks. An overview of the various network configurations tested in this scenario is illustrated in figure 4.4. In addition, we have differentiated between transit networks and tactical edge networks in table 4.6.

During the tests with limited network conditions, we have only tested our application with our own server. This is because we need to be sure that the network limitations we set actually are the bottleneck, which requires us to have full control over the network. CEI is hosted by others, and testing would then have to happen over Internet, which is out of our control. An overview of the test setup is displayed in figure 4.5.

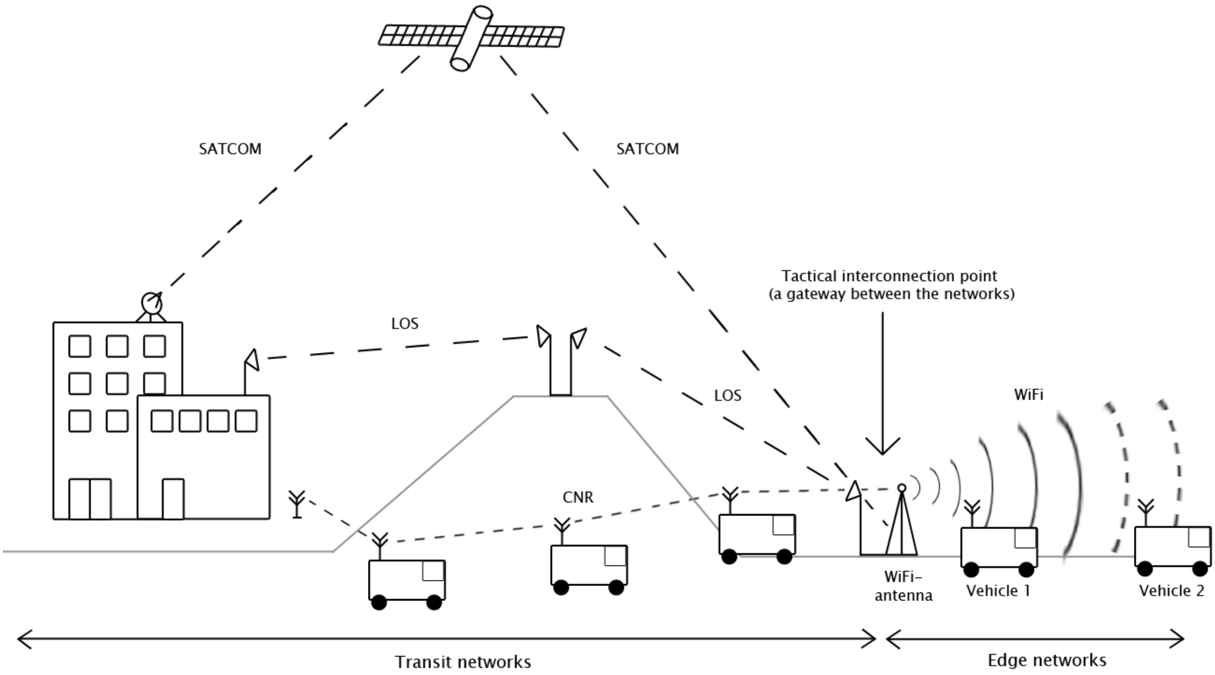


Figure 4.4: Overview of Tested Networks

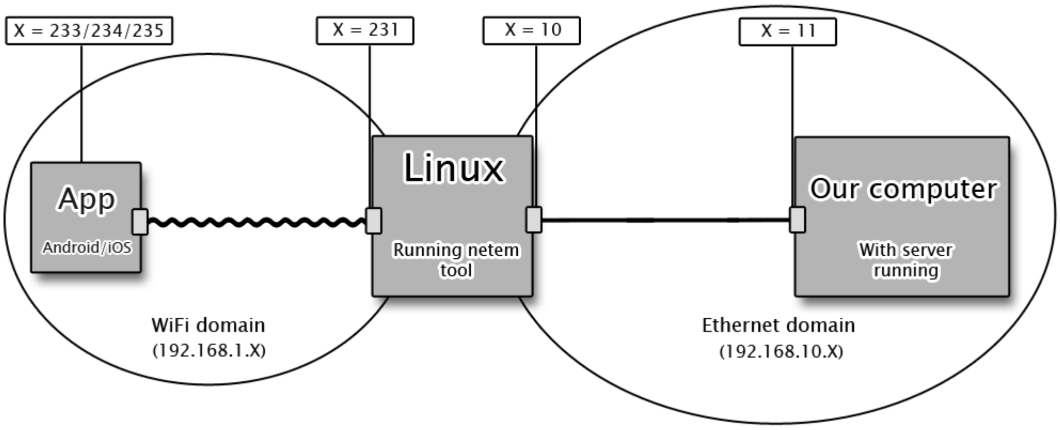


Figure 4.5: DIL Tests - Limited

Both mobile platforms, iOS and Android, and all three devices were used for testing of the cases defined below. The iOS device had some problems with setting up the desired network settings, but it worked once we included an IP address for a DNS server. The Android devices did not need this setting for our testing purposes.

### 4.5.1 Objective

The objective in the following tests is to verify that PISA manages and tolerates the different aspects of the limited DIL networks. The data rate (bandwidth) can be low, the delay high and the PER also high. These are attributes which deteriorate the probability of successful transmission of messages. PISA must then be able to store the messages and free the user to try and retransmit them later. We also want to see how many of the messages are successfully sent, and if any are lost. Once again, PISA should not crash while trying to send the position or observation. With periodic positions, PISA should count the correct amount of successful and failed positions. These tests also are performed to see how our application behaves when introduced to several limited features in the network. It should still act as described above, and in all the previous test cases earlier in this chapter.

### 4.5.2 Execution

It is again the main functionality being tested, i.e., both on demand- and periodic positions, and the different types of observations. We expect that PISA is able to successfully send the messages to the server in most of the networks. However, since some of them are more limited than others, PISA must still be able to cache unsuccessful messages when it is unable to send to server. Then the user is enabled to retransmit at a later point. PISA may store maximum one position and one observation at a time when it is unable to transmit the message, except from the periodic positions. These must be counted and presented to the user as they succeed or fail in real-time. The time it takes to send the various data initially varies (observation with or without an image), but they might differ even more between the tested networks, due to their different characteristics.

The next five sections contain information about the tested networks, and their respective results and analysis.

### 4.5.3 First Test Network - SATCOM

SATCOM is an emulated satellite link network, and it is defined as a transit network. This network has medium bandwidth, high delay and zero PER. The relevant characteristics can be found in table 4.6. SATCOM is also included in figure 4.4 as one of the illustrated networks.

### Results and Analysis

In this test network, we observed that all messages were successfully sent to the server, however sometimes very slow. The positions had some small delay, but arrived every time. It was pretty much the same with the observations including only description or description and position. They had some delay, but we experienced successful transmission in about two-five seconds. The observation containing an image in addition, were slower at times. The transmission time varied from about less than 10 seconds and up to (and sometimes over) 30 seconds, which is not optimal. This is due to the increased packet size, because an image takes more space than a little description and a pair of latitude and longitude values. When the transmission time exceeded the timeout defined by PISA, the user gets a message telling him that the observation failed, and the observation would be cached on PISA. However, the observation did actually arrive at the server, only at a later point in time.

Compared with the function test (defined in section 4.2) which execute the tests over a normal network, we understand that we may experience some more delay in this scenario. SATCOM has both lower data rate and higher delay than other ordinary networks, which could make it troublesome to send larger data packets, e.g., like observations including images.

There were no huge variations in the results between the devices. The behavior was very similar on all three devices. The results from this test are summarized in table 4.7.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Limited success | Limited success |

Table 4.7: Results of the Limited DIL Tests - SATCOM

The observations with images could take a long time to send, and could therefore be perceived as failed messages. In some cases, the user would get a message saying that the observation was not received at the server, and the data would then be stored on the device. However, the message was still in transit, and would be received at a later time. This is due to the sending time-out in PISA triggering before the message has arrived at the server. This could lead to duplicate messages on the server side (due to the user re-sending the information). Management of duplicate information on the server side is beyond the scope of this thesis, but should be fairly simple to implement due to all the messages containing unique information in the form of a sender identifier combined with a message time stamp. In order to limit the frequency of this issue arising, one could alter the message sending time-out in PISA when using very slow networks.

In these tests, all messages were received at the server, but the issue described above means that the tests relating to observations containing an image have only a limited success. The DIL tests performed in this scenario verify requirement number four in table 2.1 to a certain degree, now concerning the limited aspect of DIL. Both requirement number five and six are fully verified, since all the appropriate data are handled correctly. Since the observations containing an image had limited success, we must therefore conclude that the limited DIL test in the SATCOM network is partially successful.

### 4.5.4 Second Test Network - Line-of-Sight (LOS)

The other network type we tested over was a so-called LOS, or line-of-sight, network. This is a radio-based type of network that is distinguished by that there are no physical obstacles between the nodes in the network. This way, the nodes can see each other directly even though they may be far from one another geographically. This network type is common in military scenarios, where it is used as a transit network between deployed tactical networks and fixed infrastructure networks. A LOS network is typically established by placing nodes with directed antennas, and direct two and two antennas against each other. This way one can achieve good capacity with wireless communication over long distances.

This network has high bandwidth, low delay and zero PER. Table 4.6 defines the specific characteristics of this network. LOS is illustrated as one of the tested networks in figure 4.4.

#### Results and Analysis

In this test network, we observed again that all messages were successfully sent to server. The positions were sent very fast, and arrived every time. We experienced near no delay at all. With the observations including description, and description and position, we experienced very much the same behavior. The messages were sent and received at the server very fast, and the transmission occurred most often immediately after pressing the send-button (0.1-0.5 seconds). The observation including an image, were not considered as much slower at all. However, due to the increased size of the message, they took around a second to send, at tops.

Compared with the function test, which is defined in section 4.2, we experienced similar behavior since LOS is a very fast network with high data rate and low delay.

There were no major variations in the results between the devices in this test either. The behavior was very similar on all three devices. The results from this test are summarized in table 4.8.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Success | Success |

Table 4.8: Results of the Limited DIL Tests - LOS

The observations in this test tell us that it is successful, and our expectations were correct. All messages were successfully received at the server. The results of this DIL test verifies requirement number four in table 2.1, with respect to the limited aspect of DIL. Requirement number five and six are also verified since both loss tolerant and essential data are received by the server. We therefore conclude that the limited DIL test in the LOS network is successful.

### 4.5.5 Third Test Network - WiFi 1

WiFi 1 belongs to the edge networks, not the transit ones. In this test network, the user is presumed to be in the WiFi's "sweet spot", i.e. well inside of the connection range. WiFi 1 has high bandwidth, medium delay and low PER. The actual characteristics are defined in table 4.6. WiFi 1 is illustrated as the position vehicle 1 has in figure 4.4.

#### Results and Analysis

In the first of the WiFi networks, we observed that all messages were successfully sent to server. Positions were transmitted fast. We only noticed some slight delay. The observations including description, along with the observation with description and position, had the same behavior. These messages were sent and received at the server pretty fast, and the transmission was successful after approximately one second. The observation including an image only took a bit more time. In general, they took a about one-three seconds to send.

Compared with the function test, which is defined in section 4.2, we experienced very similar behavior, due to WiFi 1 being a type of a “normal” wireless network. It has a high data rate, medium delay and low PER. This PER is only on 1 %, so the probability of packets being corrupted or dropped is low. However, it may happen, and produce poorer results.

There were no big variations between the devices used for testing in these results either. The behavior was highly similar on all three devices. The results from this test can be found in table 4.9.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Success | Success |

Table 4.9: Results of the Limited DIL Tests - WiFi 1

The observations during this test show that this also is successful, and our expectations were correct. All messages were successfully received by the server. The results of this DIL test verify both requirement number four, five and six in table 2.1, once again concerning the limited part of DIL and that all messages are handled correctly. Our conclusion is thus that the limited DIL test in the first one of the WiFi networks is successful.

### 4.5.6 Fourth Test Network - WiFi 2

WiFi 2 is defined as an edge network, like WiFi 1 is. However, here the user is presumed to be in the WiFi’s “edge”, i.e. at the very edge of the connection range. WiFi 2 has both high bandwidth and medium delay, as WiFi 1. What separates them is that WiFi 2 has a high PER (20 %). The specific characteristics can be found in table 4.6. The position of vehicle 2 in figure 4.4 illustrates this type of network.

### Results and Analysis

In the second of the WiFi networks, we observed some mixed, however, successful results. All of the messages were successfully received at server, however not always very fast, in addition to the transmission time being very varied. Positions could be transmitted fast (about one second), or somewhat slow (five to ten seconds). The latter is considered somewhat slow, since in all earlier scenarios it always takes about less than one and up to three seconds. Ten seconds is the maximum we observed. The observations including description, along with the observation with description and position, had the same behavior. The transmission time varied similar to the one with positions. The observation including an image usually took a lot more time. They usually varied from taking 5-20 seconds, and in some cases up to around and over 1 minute. Sometimes the user could get a notification that the message failed, even though it was received at server at some later point.

Compared to the previous test, the first WiFi network, we experienced at times similar behavior, and maybe more often much longer transmission time. The second WiFi network also has a high data rate and medium delay. However, the PER is increased to a whole 20 %, which really is a quite extensive amount. The probability of packets having errors or being dropped in the previous scenario were one in a hundred, now it is one in five. This exacerbates our results, since the initial message may fail at the first try, and even on the second and third, and so on. This is what extends the transmission time.

What happens in the background is the works of the transport layer Transmission Control Protocol (TCP) [27]. TCP has something called a retransmission timeout. Once a packet has an error or is dropped, TCP tries to retransmit the package up to a certain number of times. This is what increases the transmission time of the different messages, when they fail, TCP tries to send them again. If the second try fails, TCP retransmits for you again, and so on. However, TCP eventually gives up. In the observed cases, the messages always arrived at the server, but often very late. The user would get the message telling that the data was not sent, and it is cached. This behavior could again lead to duplicate messages on the server, when the user later sends the previous data, even though it might have been successful last time.

In this case, there were once more no big variations between the devices' results. The behavior was quite similar on both the iPhone, Google Nexus and Samsung Galaxy Tab. The results from this test are summarized in table 4.10.

The observations made in this test show that it is somewhat successful. All of the messages were successfully received by the server. However, the user may get notified that the observations with image failed, when they really did not. This was due to the

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Limited success | Limited success |

Table 4.10: Results of the Limited DIL Tests - WiFi 2

fact that these messages were often very slow. The results of this DIL test verifies (to some extent) requirement number four in table 2.1, once again concerning the limited part of DIL. Requirement number five and six are verified, due to the data being handled correctly. The user would get a message saying that the observation was not received at the server in most cases, and the data would be cached on PISA. Still, the message would be received at a later time. Our conclusion is thus that since all messages are successfully received at some point in time, even though some messages being notified as unsuccessful, the limited DIL test in the second one of the WiFi networks is partially successful.

### 4.5.7 Fifth Test Network - Combat Net Radio (CNR) with Forward Error Correction (FEC)

CNR is a term that is used for a group of wireless military communication networks. The characteristics of these networks are that they have no fixed infrastructure, but they are based on the use of radio as the only communication medium. The nodes of a CNR are often mobile (vehicle mounted radios are an example), and these nodes form a communication network which forwards network traffic on behalf of each other. In a CNR there is a trade-off between possible data rate and range (long range means low data rate), which in practice means that the data rate in a CNR often is low. CNR often acts as both the edge network (in the vehicle example, the edge systems on the vehicle use CNR as edge networks) and as a transit network (in the same example, foot soldiers could use the vehicles' radios for transit).

This network has low bandwidth, medium delay and low PER. Table 4.6 provides the specific characteristics of CNR. This network is illustrated in figure 4.4, like the rest of the tested networks.

### Results and Analysis

With the CNR network, we also observed some mixed results. Yet, they were all successful results. All of the messages were successfully received at server, except that the observations containing images were often extremely slow, and seemed to not appear at the server. However, they always arrived (after some, often *long*, time). Positions were transmitted pretty fast. We only noticed some very slight delay. We got very similar results from the observations including description, and the observations with description and position. All these messages were successfully received by the server in about one to three seconds. The results looked to be similar to the ones from the first WiFi network, however a little bit slower due to very low bandwidth in comparison. The observation including an image were very slow. Minimum transmission time discovered was about one minute. This is of course because of the extremely low bandwidth in comparison to the other tested networks. Some of these messages arrived at the server near five to ten minutes after sending the message. This is not at all optimal, so it cannot be seen as a complete success. However, as previously mentioned they arrived every time, although after a very long time.

When looking at the results, we experienced similar behavior to the second WiFi network, where the observations including an image also were very slow to be successfully sent. However, the reason for this is different. In the second WiFi test, the messages were slow due to the PER. Here, it is the low data rate that is responsible. The CNR network has a low data rate and medium delay. The data rate makes the transmission of observations with images extremely slow, since the packet size is greater than with the other messages. So the low bandwidth struggles with sending the whole packet.

There were no big variations between the devices' results. The behavior was very similar on all three devices. The results from this test is summarized in table 4.11.

| **Messages** | **Result - Android** | **Result - iOS** |
|---|---|---|
| Position | Success | Success |
| Observation with description | Success | Success |
| Observation with description and position | Success | Success |
| Observation with description, position and image | Limited success | Limited success |

Table 4.11: Results of the Limited DIL Tests - CNR with FEC

The observations we made here show that again it is only somewhat successful. All messages were successfully received by the server. However, the user may be presented with a message telling that the observations with image failed, when they actually did not. This is because of the messages were often extremely slow in this case. Still, the results of this DIL test verifies requirement number four in table 2.1, when concerning the limited part of DIL, at least to some degree. Also, requirement number five and six are verified due to all messages arriving at the server after some, and in some cases very long, time. Our conclusion is therefore that since all messages are successfully received at some point in time, the limited DIL test in the CNR network is partially successful.

## 4.6 GUI Tests

Another important factor to test, is the GUI of PISA, and what we test are the user experiences of the application. In this thesis, we do not focus on performing a full-scale test of PISA's GUI. What we want to achieve with these tests are to identify aspects of the GUI which makes it possible to make a better GUI when you develop a production system with the same functionality later. Therefore, we settle for a smaller, more informal test. This is after all a small-scale iterative initial test. This is usually the first step in a larger testing, if the goal of this application was to go into production. However, PISA is only a prototype which is enabled for further development by others.

Due to time limitation, and the fact that we were only conducting a limited scale test of a prototype, we used a small group consisting of three technologically skilled test subjects to test our prototype application and provide them with a short questionnaire. The questionnaire is based on a figure 1 in [37]. This report essentially discuss and evaluates the importance of user interface aesthetics, and lists several different evaluation methods. The methodology we apply is further discussed in section 4.6.1.

The methodology is first described, before the survey questions are discussed. Later, the execution of this test is reviewed, and at last the results are presented with the analysis of what they mean. Some similarities are debated until lastly, potential improvements are discussed.

### 4.6.1 Methodology

We perform a qualitative evaluation of the user experience of PISA, since one cannot obtain representative quantitative results from such a small group. The methodology we chose to use, is a light version derived from the one called "Classical aesthetic judgement" by Lavie and Tractinsky. This subjective evaluation method can be found in table 3 in [37]. We provide the application installed on one of the devices along with the questionnaire described in the following section to the small group of testers and let them evaluate our app.

The “Classical aesthetic judgement” method includes several criteria that are suitable for our testing purposes. Important keywords when it comes to elements of evaluation in focus are: Pleasant, clear, clean and symmetric design. These aspects are much what we look for in our application. PISA is a simple app, which should be easy to use and have a clean and symmetric look. An important tool used with this method is often a questionnaire, which we too apply in this thesis. By using this and defining our own questions, we can easily control which aspects we want evaluated. The pros of this method are that it is both simple and quick, and we can score our results. Since this is an initial small-scale test, simple and quick are desired attributes.

### 4.6.2 Preview of Survey Questions

We have chosen to base our questions on figure 1 in [37]. It illustrates important aspects of user experience, such as usefulness and different aspects of usability (efficiency, controllability, helpfulness and learnability). The testers were provided with a questionnaire including a short introduction and several questions. The introduction and questions provided to the testers were as follows:

*Introduction*: The application (PISA) is a simple app that is intended for people which are out and moving in the terrain, and can report where they are located and what they observe.

1. Which device did you test on?
2. How easy was it to understand what kind of functionality the app offers?
3. How easy was it to figure out what you needed to do to configure/get started with using the application?
4. How easy was it to use the app?
5. How easy did you think it was to navigate through the app?
6. Did you get sufficient information on what happened, e.g., if an error situation occurred?
7. What did you think of the user interface?
8. Were there any functionality that you felt was missing, something that you expected to be there?
9. This is a prototype that is to be further developed. Do you have any suggestions for possible improvements?

10. Other remarks:

### 4.6.3 Execution

We executed these GUI tests because we wanted to know how our prototype appears before people who do not know about PISA and what it does from before. The questionnaire was made in advance of the testing. This consists of necessary questions for us to be able to better evaluate the GUI of PISA. We handed out our prototype app, which was installed on all three devices, along with the questions, and let the testers do the rest. The group of testers comprised three competent FFI researchers, without any prior knowledge of the application.

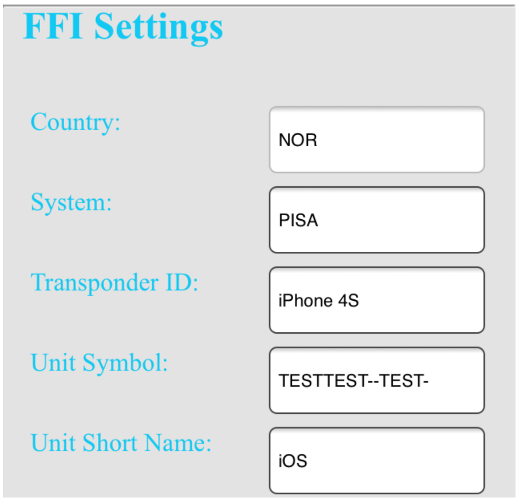
The testers did not get any more information about the app than the small introduction on the top of the questionnaire. They were left to test the functionality along with the design of PISA by themselves. However, if they had any questions, the author was there to answer these. The main focus of this test was to see how simple and intuitive the application appeared to new users. We also wanted to see if any clear differences in design or behavior between the devices were found. If something did not look or function optimally, we wanted the testers to let us know, so we could make a note of these as potential improvements.

### 4.6.4 Results and Analysis

All three devices were tested in the GUI tests. The essential feedback and the commonalities between them are discussed below. Only the highlights of the feedback are summarized here, the raw material of the questionnaire can be viewed in appendix C. Questions two through seven from section 4.6.2 are evaluated. The first and the last three of the questions are omitted here. The first is excluded because all devices were tested, and some of the testers evaluated more than one device. No one answered the last question labeled “Other remarks”, and the two last questions are further discussed in the next section (since they talk about potential improvements).

The feedback from question two was very positive. They ranged from very easy to easy to understand what the functionality of PISA was. The menu was experienced as intuitive by all testers.

The evaluation of question three was somewhat varied. The configuration part was experienced as both easy and difficult. This is due to that all devices were configured in advance of the tests. If the app had not been properly set up, the configuration part of PISA seemed somewhat difficult. There were some aspects that were not enough explained, like the FFI attributes in the settings page shown in figure 4.6. The configuration part of PISA is described in section 3.3.7. PISA could benefit from having some informative descriptions on what kind of information is supposed to be put in the various fields.



FFI Settings

Country: NOR

System: PISA

Transponder ID: iPhone 4S

Unit Symbol: TESTTEST--TEST-

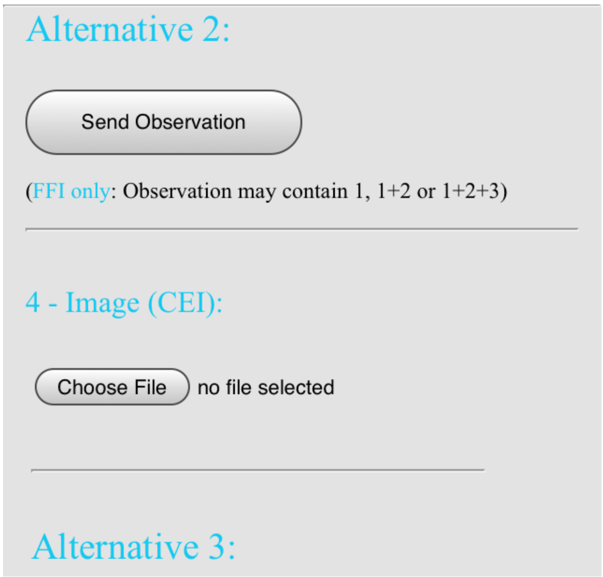
Unit Short Name: iOS

Figure 4.6: PISA Settings - FFI Attributes Screenshot

On question four, the testers answered that it was relatively easy to use the app, and that there were no huge difficulties of using it. However, the observation page was considered as a bit complicated. The different alternatives were not that easy to understand. Figure 4.7 illustrates a portion of the observation page, which could be understood as a bit cluttered and hard to understand.

With navigation in question five, the testers seemed to think it was pretty easy. However, what they all missed was a “back” button. A “menu” button is present, but some did not even see that button at first. The “menu” button should either way be easier to push (especially on the iOS device, which is the smallest), and much more visible. The same goes for the “login” button. The “menu” button could also be named “home” since it takes the user back to the starting point of the application. The two buttons “menu” and “login” are shown in figure 4.8, specified with a red outline.

When it came to if the user was provided with enough information at all times (from question six), the feedback was very positive. They turned off their network connection at the devices, and they thought the information on what happened was very good during these situations, and generally otherwise also. The only thing noted, is that the information given sometimes felt double and unnecessary with information from both pop-ups and the results text field. Figure 4.9 contains an example of the user possibly getting some redundant information.



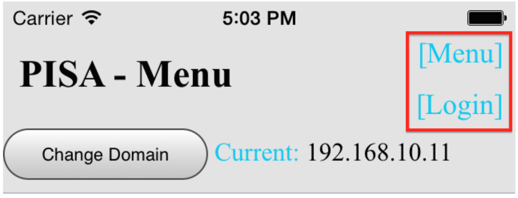
Alternative 2:

(FFI only: Observation may contain 1, 1+2 or 1+2+3)

4 - Image (CEI):

Alternative 3:

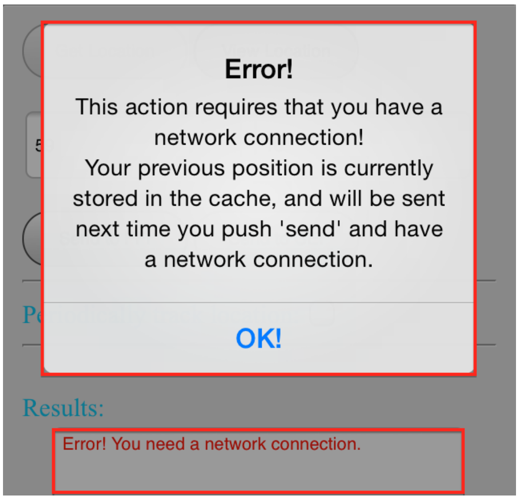
Figure 4.7: PISA Observations Screenshot



PISA - Menu

Current: 192.168.10.11

Figure 4.8: PISA Navigation Buttons Screenshot



Error!

This action requires that you have a network connection!
Your previous position is currently stored in the cache, and will be sent next time you push 'send' and have a network connection.

Error! You need a network connection.

Figure 4.9: PISA Notification Screenshot

Question seven, which asks how the tester feels about the user interface, had a lot of feedback. The GUI sometimes felt messy and chaotic, this was only the case with the observations page. It should have more defined lines, and drop the alternative "send" buttons. It should have fewer buttons, because they could be combined. The way of choosing an image also felt poor, and should be improved. It was also unclear what the buttons labeled "Send to FFI" and "Send to CEI" actually meant. An example of the observations page can be viewed in the previous figure 4.7. The two "send" buttons are displayed in figure 4.10.

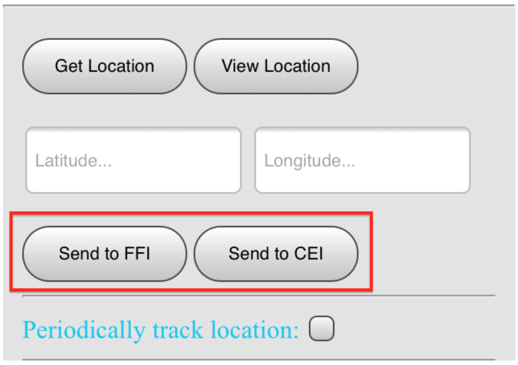


Figure 4.10: PISA Submit Buttons Screenshot

### 4.6.5 Potential Improvements

In this section, potential improvements based on the results from question eight and nine from section 4.6.2 are proposed in the following list:

- Menu/login buttons should be enlarged and moved further away from each other.
- The "Settings" button should not be equivalent to the report buttons. It is important to distinguish between functionality and configurations.
- The observations page was perceived as a bit messy. Buttons could be combined, and automatically send correct alternative of observation according to which fields are that are filled in.
- The application should also have a "back" button.
- The prototype could be expanded to have video and audio functionality.
- If the app does not really do that much when the network is down, the user could be notified when he opens PISA that the network is down and that he should try again later, before the app closes.
- The "results" fields should have another color (e.g., gray), since it seems that it is possible for the user to edit these fields.
- The "results" field could also be dropped, since the pop-ups describe the same information.
- The settings page of PISA should be more informative, and have possible explanations of what the different terms mean.
- The observation page is very long. It should be much shorter, or have the possibility of a "scroll to top" button.

## 4.7 Summary

This chapter provided the evaluation aspect of our our thesis. Both the performance and GUI of PISA were evaluated in this chapter. The evaluation tools were first provided, and it comprised of both emulators and real devices. One iOS device, and two Android devices were applied. We wanted to test different OS's and device types, along with various screen sizes and resolutions, manufacturers, and OS versions.

The first type of test that was performed was the function tests. These tests included testing of the functionality without any network limitations of any kind. This was to see that the functionality at least worked out of the box. These tests were successful. The second test we evaluated was the disconnected part of the DIL test. This was the scenario where the connection is lost. The results here were successful. The third test was about the intermittent part of DIL, where the connection is lost, but then reestablished at a later point. These tests also yielded successful results.

The next test type was the limited DIL tests, which contained multiple subtests. This was due to the limited-aspect representing different features like bandwidth, delay and PER. Therefore, five tests were carried out in five various emulated networks with distinct values of these factors. They contained a combination of either high, medium or low values of the different features. The results throughout these tests ranged from having complete success or limited success. Still, every test always resulted in some degree of success. Finally, there were no major differences in any of the results from all performed tests between the OS's, nor the three different devices.

We also performed testing of the GUI of PISA. The carried out testing followed a certain methodology, and the test subjects received a questionnaire before evaluating our application. The results of this test yielded both positive and good results, in addition to several suggestions of improvements. Most of the constructive feedback we got, we were already aware of could use more work or some modifications. However, they are essential to keep for future work on this prototype.

In this chapter, requirement number four through six were fulfilled by testing. Requirement number eight was partially satisfied through testing. The requirement about PISA needing to support and work in DIL environments was fulfilled by receiving either limited success or success from all the various DIL tests in this chapter, both the disconnected, intermittent and limited ones. The two requirements concerning that some specific messages tolerates loss and some must always arrive, are both fulfilled due to all tests (the function and DIL ones) proving that the appropriate messages always were handled correctly. The periodic positions were lost, and all other messages were cached. The requirement about having a simple and intuitive GUI was satisfied to a certain degree. In some aspects, PISA was viewed as both easy to use and understand. In some other areas, PISA was perceived as somewhat difficult, like the page where you report observations. This was anticipated, since the GUI did not get as much attention as wanted due to the time limitations. However, this requirement did not have the highest priority, only the second one.

The next, and final, chapter concludes this thesis and discusses potential future work.

# Chapter 5

# Conclusion and Future Work

This chapter contains both the conclusion of this thesis and suggestions of potential future work. The conclusion summarizes the most essential aspects throughout the thesis such as the objective, the premises and requirements, and concludes if these were all fulfilled or not. Future work discusses the work left for further research that was omitted from this thesis due to the time limitations.

## 5.1 Conclusion

The objective of this thesis was to develop a handheld prototype application for reporting positions and observations. A civilian COTS solution targeted at military use, and with a special focus on the tactical edge and DIL environments, is preferred due to the low cost. Using COTS devices such as smartphones and tablets instead of military hardware, is usually much less expensive. Still, this kind of devices serve as very powerful sensor platforms. The application needs to be platform independent to avoid placing restrictions on use. It is at this point we differ from previous work, as these focus on a specific platform.

The requirements and premises for the prototype solution were defined in table 2.1. In the requirements analysis, we reached the following aspects which needed to be fulfilled:

1. *Premise* - Application must be platform independent.
2. *Premise* - Application must support major smart device OS's (iOS and Android).
3. *Premise* - Application must be implemented using a free and open source framework.
4. Application must support, and work in, DIL environments (e.g., tactical edge).
5. Application must support loss tolerant messages, i.e., positions (these messages can be lost).

6. Application must support messages that are not tolerant of loss, i.e., observations (these messages must arrive).
7. Application must use REST as communication method.
8. Application needs to have a simple and intuitive Graphical User Interface (GUI).
9. Use standards when possible.
10. Facilitate interoperability towards back-end infrastructures (SOAP Web services, NII, etc.).

The three first requirements, which are actually premises, are all fulfilled. The application *is* platform independent, and it supports, and is used on, both the iOS and Android mobile platforms. The application was implemented using a free and open source framework, PhoneGap, which facilitates for platform independent development on OS's like Android, iOS, Windows Phone and several others.

Requirement number four, five and six were shown to be fulfilled through the evaluation of PISA in chapter 4. The application worked in all the different test scenarios, both the function tests and the DIL tests. In all DIL environments, PISA achieved results that revealed either full or limited success. Both loss tolerant messages and messages that always needed to arrive at the server were at all times handled correctly. Periodic positions were lost, and all other messages were cached on the device for later retransmission if a connection error occurred. The GUI was subjectively evaluated by a small group of users. It was evaluated to be quite simple and intuitive in some areas, but also complex or tricky in other aspects. In the aftermath of the GUI evaluation, it must be acknowledged that there is room for improvement of the GUI.

The rest of the requirements, requirement number seven, nine and ten, were all fulfilled "by design" during the design and implementation of PISA in chapter 3. REST was applied as the communication method used between PISA and our two servers. Standards were used when possible. The NATO standard, NFFI, was used as the format for reporting positions to our server. FFI-incident is a proprietary format, but it serves our purpose because it was developed specifically for reporting XML-formatted observations. Reporting data to CEI is done with the JSON format, and only proprietary representations of it. However, standards are used *when possible*, so this requirement is also satisfied. Our wrapper server uses both the XML and NFFI format, so it is therefore facilitated for later interoperability towards Web services infrastructures, such as NII.

All in all, the objective of this thesis was reached. All requirements are fully satisfied, except for requirement number eight, which is partially satisfied. However, the work has thus far met the requirements, and the goal of creating a prototype is reached.

## 5.2 Future Work

The GUI was one of the areas of this thesis where a need for improvement was identified. An initial small-scale GUI test was executed with a little team of technologically skilled subjects. The first version of the GUI proved in testing to have some problems that allow for improvement. This is natural to look at as future work. The development of the GUI should optimally have followed an iterative plan, and received feedback from the testers on a periodic basis. The group of testers should also be consisting of operative personnel.

Although we have mentioned the possible importance of battery optimization in section 2.3, we did not have the time nor means to evaluate battery usage in our thesis. It should be noted that battery life on handheld devices could be a very essential factor. When the device experiences intensive or high usage, the battery could be depleted in a short amount of time, if the possibilities of charging is scarce. It is not necessarily trivial to limit or optimize battery performance with an abstraction framework like PhoneGap. One thing we could have done is to have one shorter, and more frequent, time interval for periodic positioning when the device is in charging mode, and one larger interval for when the device is in battery mode. This way, it could save some battery when sending periodic positions running only on battery.

Security is another very important factor when it comes to military settings. In our thesis, this was out of scope. We only used existing security mechanisms such as the authentication in CEI, where a username and password were required to communicate with the CEI server. The communication with our wrapper server was not secured in any way. We have focused on other aspects of our prototype solution. Still, we understand that security is a requirement in most military systems. A suggestion could be to look at solutions such as OAuth. OAuth is an open standard for authorization.

In our solution we have implemented one application, PISA, one server, the wrapper server, in addition to have integrated with an external server, the CEI server. With PISA and the wrapper server, which *we* have implemented, we have done our effort to try and facilitate for later interoperability with SOAP-based Web services infrastructures, e.g., NII. Our server functions as a wrapper to these back-end infrastructures, but the actual connection towards NII is not implemented in this thesis. We have only facilitated for this opportunity. For instance, FFI's INI-lab is an example of a solution that could be connected with our prototype app and server. Other examples could be NATO-systems like JOCWatch. A complete wrapper must be implemented for interoperability with NII using SOAP Web services for back-end communication.

Automatic retransmissions of messages could be desirable in later improvements of our application. We chose to not utilize this opportunity to save network resources, since

PISA is aimed to work in DIL environments with various network limitations. To implement this functionality is not trivial to achieve either.

Formats like NFFI and FFI-incident are utilized in our prototype. The data format of XML is also used between PISA and our server. XML is an official data format standard and NFFI is a NATO standard, which both enhances the ability of interoperability between solutions. JSON is used as the data format in CEI's prototype system, and only proprietary representations are utilized. A way to make interoperability easier, would be to migrate CEI towards NATO standards.

In the evaluation of PISA used in DIL environments, we experienced a recurrent issue with messages containing big data, i.e., the observations including an image, being delivered to the server. PISA could report that these messages were not delivered, due to high delay in the SATCOM network, high PER in the second WiFi network or low data rate in the CNR. Still, these messages were actually received at the server, but at times much later (up to several minutes in the worst cases). It could be wise to let the user specify an application-side timeout in the settings/configuration of PISA. Another possibility could be to expand PISA in a way that it itself tries to adjust the timeout based on the type of network it is currently connected to.

PISA has a focus on being a mobile reporting application. For PISA to be a complete situational awareness tool, it should employ a two-way communication, instead of the unidirectional communication it utilizes as of now. Our application could for instance be expanded to receive data from CEI, in addition to just send to CEI. The received messages from CEI could be integrated with a map to display the reported positions and observations. PISA could also be extended to receive positions and observations from *other* users, to benefit from observed incidents and locations of other units in the area.

# Bibliography

[1] P. Bartolomasi, T. Buckman, A. Campbell, J. Grainger, J. Mahaffey, R. Marchand, O. Kruidhof, C. Shawcross, and K. Veum. NATO Network Enabled Capability Feasibility Study. NNEC Feasibility Study, Brussels, Belgium, October 2005.

[2] CloudPact. Mowbly Website. `http://www.mowbly.com/`, accessed 2014-07-30.

[3] M. Crewes, M. Asche, G. Singh, and J. H. Gibson. SPARCCS – Smartphone-Assisted Readiness, Command and Control System. 17th International Command and Control Research and Technology Symposium (ICCRTS) - "Operationalizing C2 Agility", Fairfax, VA, USA, 2012.

[4] F. Dandashi, J. Higginson, J. Hughes, W. Narvaez, M. Sabbouh, S. Semy, and B. Yost. Tactical Edge Characterization Framework. MITRE Technical Report MTR070331, McLean, VA, USA, 2007.

[5] P. J. Denning. Computer Science: The Discipline. In A. Ralston and D. Hemmendinger (eds.), 2000 Edition of the Encyclopedia of Computer Science, 2000.

[6] D. O. Eggum. Efficient SOAP messaging for Android. Master's thesis, University of Oslo, Department of Informatics, May 2014.

[7] J. B. Evans, B. J. Ewy, M. T. Swink, S. G. Pennington, D. J. Siquieros, and S. L. Earp. TIGR: The Tactical Ground Reporting System. Institute of Electrical and Electronics Engineers (IEEE) Communications Magazine, USA, 2013.

[8] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, 2000.

[9] E. Gjørven, F. T. Johnsen, A. Fongen, T. H. Bloebaum, and B. K. Reitan. Towards NNEC: Breaking the interaction barrier with communication and collaboration services. FFI technical report 00943, Kjeller, Norway, 2014.

[10] L. H. Karlsen and B. K. Reitan. CEI - et sosialt taktisk rapporteringssystem: Teknisk beskrivelse av Android klient for smarttelefon og nettbrettstøttet til CEI-systemet. FFI technical report 00526, Kjeller, Norway, 2014.

[11] K. Lund, F. T. Johnsen, T. H. Bloebaum, and E. Skjervold. SOA Pilot 2011 - service Infrastructure. FFI technical report 02235, Kjeller, Norway, 2012.

[12] Adobe Systems Inc. Phonegap - Supported Features. `http://phonegap.com/about/feature/`, accessed 2014-07-30.

[13] Adobe Systems Inc. Phonegap Documentation - Platform Support. `http://docs.phonegap.com/en/3.3.0/guide_support_index.md.html#Platform%20Support`, accessed 2014-07-30.

[14] Adobe Systems Inc. Phonegap Website. `http://phonegap.com/`, accessed 2014-07-30.

[15] Adobe Systems Inc. Platform Guides. `http://docs.phonegap.com/en/edge/guide_platforms_index.md.html`, accessed 2014-07-30.

[16] Apple Inc. Apple Developer Downloads. `https://developer.apple.com/downloads/index.action`, accessed 2014-07-30.

[17] Apple Inc. iOS Developer Program. `https://developer.apple.com/programs/ios/`, accessed 2014-07-30.

[18] Apple Inc. Xcode. `https://developer.apple.com/xcode/downloads/`, accessed 2014-07-30.

[19] Ecma International. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, accessed 2014-07-30.

[20] FeedHenry Ltd. FeedHenry Website. `http://www.feedhenry.com/`, accessed 2014-07-30.

[21] Google. Android SDK. `http://developer.android.com/sdk/index.html`, accessed 2014-07-30.

[22] Google. Setting Up the ADT Bundle. `http://developer.android.com/sdk/installing/bundle.html`, accessed 2014-07-30.

[23] Google. Using Hardware Devices. `http://developer.android.com/tools/device.html`, accessed 2014-07-30.

[24] Internet Engineering Task Force (IETF). RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, accessed 2014-07-27.

[25] Internet Engineering Task Force (IETF). RFC 4287 - The Atom Syndication Format. `http://tools.ietf.org/html/rfc4287`, accessed 2014-07-28.

[26] Internet Engineering Task Force (IETF). RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format. `http://tools.ietf.org/html/rfc7159`, accessed 2014-07-28.

[27] Internet Engineering Task Force (IETF). RFC 793 - Transmission Control Protocol. `http://www.ietf.org/rfc/rfc793.txt`, accessed 2014-07-25.

[28] Joyent Inc. Node.js. `http://nodejs.org/`, accessed 2014-07-30.

[29] Kony Inc. Kony Website. `http://www.kony.com/`, accessed 2014-07-30.

[30] Motorola Solutions Inc. Rhodes Website. `http://www.motorolasolutions.com/US-EN/RhoMobile+Suite/Rhodes`, accessed 2014-07-30.

[31] Software AG. webMethods Mobile Designer Website. `http://www.softwareag.com/corporate/products/wm/integration/products/mobile/overview/default.asp`, accessed 2014-07-30.

[32] World Wide Web Consortium (W3C). CSS Specifications. `http://www.w3.org/Style/CSS/current-work`, accessed 2014-07-30.

[33] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/REC-xml/`, accessed 2014-07-30.

[34] World Wide Web Consortium (W3C). HTML5 - A vocabulary and associated APIs for HTML and XHTML. `http://www.w3.org/TR/html5/`, accessed 2014-07-30.

[35] World Wide Web Consortium (W3C). Web Storage - W3C Recommendation 30 July 2013. `http://www.w3.org/TR/webstorage/`, accessed 2014-07-25.

[36] Xamarin Inc. Mono Website. `http://www.mono-project.com/Main_Page`, accessed 2014-07-30.

[37] M. Pajusalu. The Evaluation of User Interface Aesthetics. Master's thesis, Tallinn University, Institute of Informatics, Spring 2012.

[38] R. Porta. Friendly Force Information Sharing — Lessons Learned and way towards NNEC. Presentation at the 7th NATO CIS Symposium, Prague, Czech Republic, October 2008.

[39] B. K. Reitan. Discussion around CEI and their experiences. "Personal Communication", Kjeller, Norway, 2014.

[40] B. K. Reitan, M. Fidjeland, H. Hafnor, and R. Darisiro. Approaching the mobile complex - In search of new ways of doing things. 17th International Command and Control Research and Technology Symposium (ICCRTS) - "Operationalizing C2 Agility", Fairfax, VA, USA, 2012.

[41] K. Scott, T. Refaei, N. Trivedi, J. Trinh, and J. P. Macker. Robust Communications for Disconnected, Intermittent, Low-Bandwidth (DIL) Environments. IEEE Military Communications Conference (MILCOM), Baltimore, MD, USA, 2011.

[42] S. Simanta, D. Plakosh, and E. Morris. Web services for handheld tactical systems. IEEE International Systems Conference (SysCon), Montreal, QC, Canada, April 2011.

[43] M. Skjegstad, F. T. Johnsen, and J. Nordmoen. An Emulated Test Framework for Service Discovery and MANET Research based on ns-3. 5th International Federation for Information Processing (IFIP) International Conference on New Technologies, Mobility and Security (NTMS), Istanbul, Turkey, 2012.

[44] J. Sonnenberg. Disconnected, Intermittent, Limited (DIL) Communications Management Technical Pattern. Network Centric Operations Industry Consortium (NCOIC) DIL, USA, 2011.

[45] A. Welin, K. Johannessen, S. Olimstad, M. A. Krog, and L. Sandvik. Cross-Platform Mobile Development. Bachelor's thesis, Oslo and Akershus University College of Applied Sciences, May 2012.

# Nomenclature

| | |
|---|---|
| ADT | Android Developer Tools |
| AMQP | Advanced Message Queuing Protocol |
| API | Application Programming Interface |
| App | Application |
| AVD | Android Virtual Device |
| C2 | Command and Control |
| CEI | Collective Environment Interpretation |
| CLI | Command-Line Interface |
| CNR | Combat Net Radio |
| CORBA | Common Object Request Broker Architecture |
| COTS | Commercial off-the-shelf |
| CSS | Cascading Style Sheets |
| DIL | Disconnected, Intermittent, Limited |
| FEC | Forward Error Correction |
| FFI | Norwegian Defence Research Establishment<br>NOR: Forsvarets Forskningsinstitutt |
| GPS | Global Positioning System |
| GUI | Graphical User Interface |
| HTML | HyperText Markup Language |

| | |
|---|---|
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| ID | Identification/Identifier |
| IDE | Integrated Development Environment |
| JSON | JavaScript Object Notation |
| LAN | Local Area Network |
| LOS | Line of Sight |
| NATO | North Atlantic Treaty Organization |
| NFFI | NATO Friendly Force Information |
| NII | Networking and Information Infrastructure |
| NNEC | NATO Network Enabled Capability |
| OS | Operating System |
| PER | Packet Error Rate |
| PISA | The Platform Independent Sensor Application |
| REST | Representational State Transfer |
| RPC | Remote Procedure Call |
| SATCOM | Satellite Communications |
| SDK | Software Development Kit |
| SOA | Service Oriented Architecture |
| SOAP | Originally: Simple Object Access Protocol |
| SPARCCS | Smartphone-Assisted Readiness, Command and Control System |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| UI | User Interface |

| | |
|---|---|
| UNIK | University Graduate Center<br>NOR: Universitetssenteret på Kjeller |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| WIFI | Wireless Fidelity |
| WP7 | Windows Phone 7 |
| XML | Extensible Markup Language |

This page is intentionally left blank.

# Appendices

# Appendix A

# Practical Guidelines for Using PhoneGap

## General Instructions

This section contains general guidelines that are applicable for installing and using PhoneGap with both Android and iOS. For all complete platform guides, refer to the PhoneGap documentation [15]. It should be noted that this walkthrough focuses on, and is designed especially for iOS- and Mac users. Explore the documentation [15] for setup and configuration with Windows.

### Download and Install

**Step 1** Download and install Node.js [28]. This is required for the PhoneGap Command-Line Interface (CLI).

**Step 2** Once Node.js is installed, install PhoneGap by opening your command line and run the following:

```
$ sudo npm install -g phonegap.
```

### Create and Build

This section provides general instructions on how to create new projects and building them for the appropriate platforms with the command-line interface, i.e. by using the command line (**Terminal**). How to push the application to a device or emulator without using their respective IDE's is also explained.

**Step 1** In your source-code directory, type the following to create a new project called HelloWorld:

```
$ phonegap create hello com.example.hello HelloWorld
```

**Step 2** All subsequent commands need to be run within the project's directory, or any subdirectories within its scope. So, next step is to go into the project's directory by entering:

```
$ cd hello
```

**Step 3** To compile an application for a platform (say, in this case, iOS), you can type:

```
$ phonegap build ios
```

**Step 4** To test the application, you have to install it onto a device or emulator. You do this by writing:

```
$ phonegap install ios
```

**Step 5** You can also perform the *build* and *install* operations in one step. You do this by typing the *run* command, like this:

```
$ phonegap run ios
```

(If you run this command, you can skip step 3 and 4.)

## Platform-Specific Instructions - iOS

As the title suggests, the following section provides platform-specific instructions on what to download, and how to configure and use PhoneGap with iOS.

### System Requirements

- To build applications for iOS, you are required to use an OS X operating system limited to Intel-based Macs.
- The minimum required version of the IDE, Xcode, is 4.5.
- Xcode runs only on OS X version 10.7 (Lion) or greater, and includes the iOS 6 SDK.
- To submit apps to the Apple App Store, you are required to use the latest version of the Apple tools.
- If you want to test on device, iOS version 5.x is at least required.
- To be able to install apps onto a device, you must be a member of Apple's iOS Developer Program [17], which costs $99 per year.
- To deploy apps to the iOS emulator. you don't need to be registered with the developer program.

## Download and Install

**Step 1** To install the SDK, you simply need to download Xcode. There are two ways of doing this:

**a** Download from the App Store, available by searching for “Xcode” in the App Store application.

**b** Download from Apple Developer Downloads [16], which requires an Apple Developer account.

**Step 2** Once Xcode is installed, you might need to install several command-line tools. The tools might already be installed, depending on which version of Xcode and PhoneGap you use. To check if you have them or not, in Xcode, go to **Xcode** → **Preferences**, then click the **Downloads** tab. From the **Components** panel, see if **Command Line Tools** is listed. If it is listed, press the **Install** button next to it. If not, it is already installed.

Another way to check, is to open a **Terminal** window and start to type *xcode* and press the tab key. If *xcode-select* and *xcodebuild* appear in the results, the command-line tools are installed.

## Create and Build

**Step 1** To set up a new project (in your source-code directory), you can type:

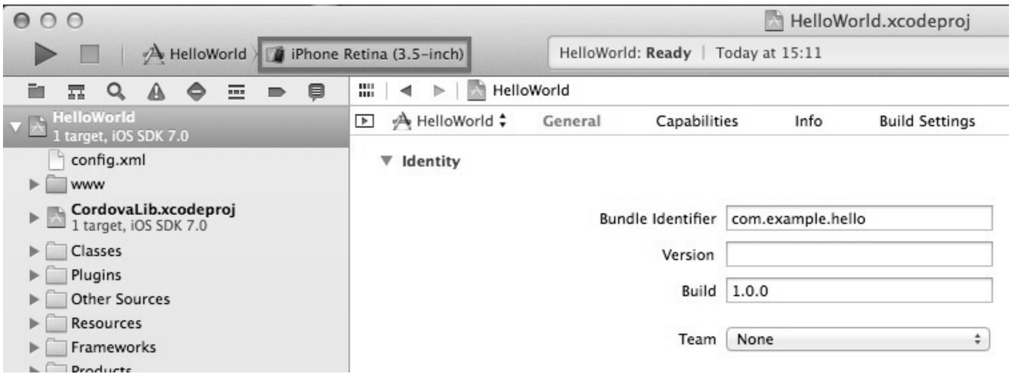
```
$ phonegap create hello com.example.hello "HelloWorld"
$ cd hello
$ phonegap build ios
```

**Step 2** Once created, you open the project in **Xcode** by double-clicking the *hello/platforms/ios/hello.xcodeproj* file.

## Deploy to Emulator

**Step 1** Make sure that the desired project is selected, then select the intended device from the toolbar’s **Scheme** menu, such as the iPhone Retina (3.5-inch) emulator as highlighted in figure A.1.

**Step 2** Press the **Run** button to the left of the **Scheme**. This builds, deploys and runs the application in the emulator.



Bundle Identifier: com.example.hello
Version:
Build: 1.0.0
Team: None

Figure A.1: Emulator Scheme in Xcode

### Deploy to Device

**Step 1** To be able to deploy to an iOS device, you have to be a paying member of the Apple iOS Developer Program as mentioned in the system requirements. In addition, you have to create a *Provisioning Profile* along with some other setup. Information on this can be found on Apple's Developer Portal [17].

**Step 2** Plug the device into your Mac by using the USB cable.

**Step 3** Select the correct project in the **Scheme** drop-down list.

**Step 4** Select your device from the **Device** drop-down list, instead of one of the emulators.

**Step 5** Press the **Run** button to build, deploy and run the application on your device.

## Platform-Specific Instructions - Android

As the title suggests, the following section provides platform-specific instructions on what to download, and how to configure and use PhoneGap with Android.

### System Requirements

- For system requirements, explore the Android Developers website, under the Android SDK section [21].
- Cordova supports Android 2.2, 2.3 and 4.x. Android Jelly Bean (4.3) is the OS being tested in this thesis.

- Unlike iOS, developing and installing apps onto an Android device is completely free of charge.

## Download and Install

**Step 1** Install the Android SDK from [21]. Here you can download the ADT bundle which includes a version of the Eclipse IDE, in addition to the essential Android SDK components.

**Step 2** On Windows, the ADT bundle is packaged with an installer. On OSX and Linux, you simply unpack the downloaded bundle in a preferred location (one where you store development tools is suggested).

**Step 3** For more information on how to set up the Android SDK, see [22].

**Step 4** For PhoneGaps command-line tools to work, it is necessary to include the SDK's *tools* and *platform-tools* directories in your PATH environment. On Mac, you can use a text editor (like TextEdit or vi) to create or modify the *~/.bash_profile* file. You can then add a line such as the following, depending on the location and *name* of the SDK:

```
$ export PATH=$PATH:/Development/adt-bundle/sdk/platform-tools:
/Development/adt-bundle/sdk/tools
```

## Create and Build

**Step 1** To set up a new project (in your source-code directory), you can type:

```
$ phonegap create hello com.example.hello "HelloWorld"
$ cd hello
$ phonegap build android
```

**Step 2** Once created, you can launch the **Eclipse** application, select the **New Project** menu item and further choose **Android Project from Existing Code** and press **Next**. Here you navigate to *hello*, or whichever directory you created for the project, and then to the *platforms/android* subdirectory. Finally, press **Finish**.

## Deploy to Emulator

You can use the PhoneGap utility to run an app in an emulator, or you can run it within the SDK. Either way, the SDK must be configured correctly to display at least one device. To do this, you must use the **Android SDK Manager**.

**Step 1** Open the Android SDK Manager which displays various runtime libraries. Do this by either:

**a** Running *android* on the command line.

**b** From within Eclipse, press the toolbar menu button up left.

**Step 2** Then, choose **Tools** → **Manage AVDs**. In the resulting dialog, you may choose any device from the **Device Definitions** tab.

**Step 3** Press **Create AVD**, optionally modifying the name, then press **OK** to accept the changes. You can also change which API Level you want the emulator to target.

**Step 4** The AVD then appears in the **Android Virtual Devices** list.

**Step 5** To open the emulator as a separate application, select the preferred AVD and press **Start**.

**Step 6** If you work inside Eclipse, you can start the emulator by right-clicking the project and choosing **Run As** → **Android Application**.

## Deploy to Device

**Step 1** To push an app directly to a device, you need to enable USB debugging on the device. This is explained on the Android Developer site [23]. You must use a mini USB cable to plug it into your system.

**Step 2** You can deploy the app to a device within Eclipse, by right-clicking the project and choosing **Run As** → **Android Application**.

# Appendix B

# Interfacing with the CEI Server

## Introduction

CEI is previously explained and discussed in section 2.2. However, in this appendix we take a closer look at the technical details of CEI, and especially their server component. First we discuss the prerequisites and necessary steps to consider before communicating with the server. Later we describe the available interfaces/services provided by CEI before we go further into the details about using them.

In short, CEI is a "social tactical reporting system". It consists of a mobile application, a server, a web application and a small scripting language. The CEI-application is a map application intended for use with smartphones and tablets. The main functionality of the application is to let users share their *positions* and *observations* in a uniform way to all users of the CEI-service. The CEI server handles all requests, i.e., registered positions and observations and makes them available on the web application. The server also sends data back to the mobile application/web application if requested by the user.

## Connection

To be able to communicate with the CEI server, it is essential that we have a registered user in the CEI system. This must be done prior to this step, and can only be done by contacting one of the authors behind CEI [10]. Then, by clicking on the "Login" button in our PhoneGap application, PISA, an in-app browser opens and directs the user to CEI's login page as shown in figure B.1.

To proceed from this point, it is necessary to fill out the form with valid credentials, i.e., username (brukernavn) and password (passord), and log in. An example of an appropriate username and password is shown in figure B.2.

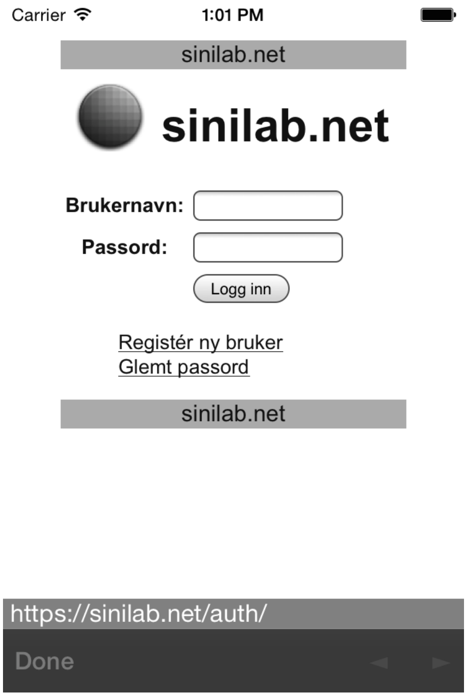
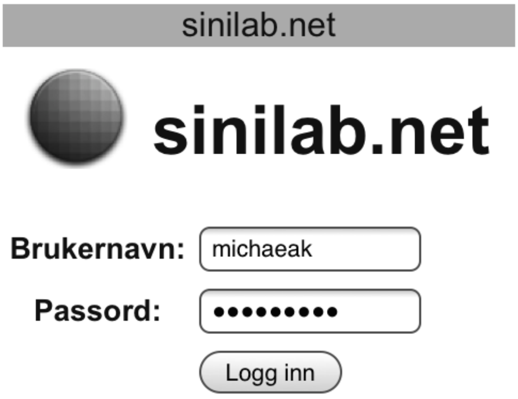


Figure B.1: iOS Screenshot from CEI's Login Page

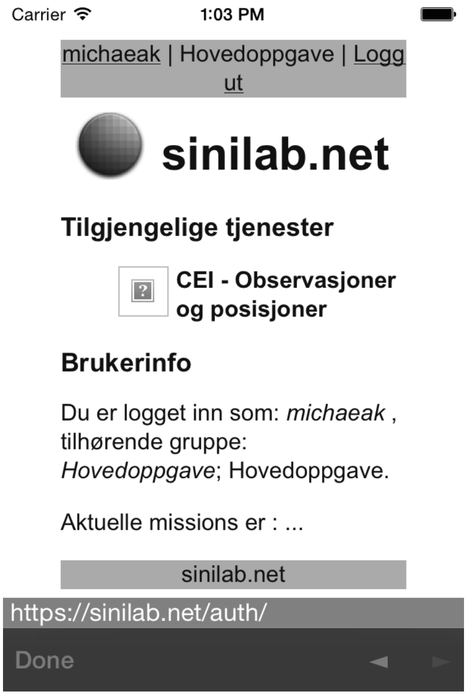


Brukernavn: michaeak

Passord: ●●●●●●●●●●

Figure B.2: CEI's Login Page - Example with Credentials

After this is done, and the user/password combination is authorized, you are logged in to the CEI system with your user. If all goes according to plan, the user is presented with a window shown in figure B.3. This page presents the user with its available services (tilgjengelige tjenester) and user info (brukerinfo). This is all we have to do to set up a proper connection between our app and the CEI server.



michaeak | Hovedoppgave | Logg ut

**sinilab.net**

**Tilgjengelige tjenester**

**CEI - Observasjoner og posisjoner**

**Brukerinfo**

Du er logget inn som: *michaeak* , tilhørende gruppe: *Hovedoppgave*; Hovedoppgave.

Aktuelle missions er : ...

sinilab.net

https://sinilab.net/auth/

Figure B.3: CEI's User Page - After Successful Login

At any time you can click on “Done” and return to PISA. After successful login you are also able to log out from the upper right corner of the user page. You are redirected to the login page again, with a red message stating that “You have logged out”. But beware, to be able to communicate with the CEI server at all times, you are required to be logged in through this interface. The CEI system operates with session based authentication for this service, so you are not logged out unless you completely delete or reset the application, or choose to explicitly log out from CEI.

## Interface

CEI provides access to the user by offering REST interfaces to communicate with their server. The main functionality of CEI's data is divided into two categories: *Positions* and *observations*. Positions contain the latitude and longitude of the user's current location. Observations are reported incidents by a user, and can include information such as a description, latitude and longitude, and optionally a picture of the incident. As earlier stated, the following interfaces all require that the user is already logged in through their login page (see figure B.1). This process is explained in the previous section. The next two sections describe each of the two main data provided by CEI, namely the position and the observation. They are here described in a more technical sense.

## Position

Positions are one of the two main data groups in CEI. It includes interfaces for registering positions, retrieving a single or all positions, in addition to a few others. These are all explained in the following sections. However, the main interface used in this thesis is the registration interface, where we report positions to the CEI server. This is the one we go into more details about, explaining some of the more technical aspects.

### Registration

To register new positions we need to have a target url to submit our data to, i.e., where the server and correct interface is located. For positions we use `https://sinilab.net/pos/register/`. To be able to publish valid data, the minimum we need to send is the *latitude* and the *longitude*. “Latitude” and “longitude” together is a geographic location of the user. We also use the variables *time* and *my_now*. “Time” is the timestamp where we first try to send the position to server. “My_now” is a timestamp of the position when it is actually sent to the server. These two can be identical, however, they can also differ. It is used to monitor if the package remained on the application for a while until it was sent to the server, e.g., due to communication disruptions.

CEI accepts requests built in a JSON format, and listing B.1 illustrates a definition of a simple JSON object for a position to be sent to the CEI server. This object contains the necessary data to be submitted to the server. It is sent using the REST interface and POST as the HTTP method. If something goes wrong, the server responds with one of the HTTP status codes listed in table B.1. If all goes well, and the position is accepted, the server responds with a status code listed in table B.2. The positions in this thesis are only sent in this specified JSON format. However, it can be sent either manually or on a periodic basis.

```
var posData = {
  'gps_data':
  {
    'position':
    {
      'longitude': longitude,
      'latitude': latitude,
    }
  },
  'time': reportedTime,
  'my_now': sentTime
};
```

Listing B.1: The Definition of a Position JSON Object

| **HTTP Status Code** | **Message** | **Description** |
|---|---|---|
| 400 | Bad Request | There is an error with the data being sent. |
| 401 | Authorization Required | The user must be logged in through CEI's login page (their /auth/ interface). |
| 500 | Internal Server Error | There is an error with the server, or the data being sent. |

Table B.1: HTTP Status Codes - Bad

| **HTTP Status Code** | **Message** | **Description** |
|---|---|---|
| 200 | OK | A position or observation was successfully sent to the server. |
| 201 | Created | A position was successfully created. |
| 302 | Found | An observation was successfully created (found) and redirected to the correct observation. |

Table B.2: HTTP Status Codes - Good

The actual data that is being sent to the CEI server when you choose to send a position to CEI, can be seen in listing B.2. The listing only shows an example of a position. The latitude and longitude have values that point to a position in Oslo, Norway. The timestamps after the “time” and “my_now” variables are Unix timestamps represented in epoch time. The value of both timestamps converts to Wednesday, 21 May 2014, 17:38:48.

```
{"gps_data":{"position":{"longitude":10.6499922517,"latitude"
    :59.9399981075}},"time":1400686728,"my_now":1400686728}
```

Listing B.2: Example of a Position JSON Object

### Other Interfaces

CEI offers additional functionality when it comes to positions. They are listed below in table B.3, but not described in detail due to the fact that they are not in this thesis' scope.

| **Interface** | **HTTP Method** | **Description** |
|---|---|---|
| Latest | GET | An interface that retrieves the latest position of the registered user. |
| All | GET | An interface that retrieves all the registered positions. |
| Tracks | GET | An interface that retrieves tracks, i.e., track history. |
| GetTime | GET | An interface that retrieves the server-time in epoch. |

Table B.3: Other Interfaces for Positions

## Observation

Observations are the second of the two main data groups in CEI. This data group has interfaces for registering observations, obtaining either a single (by ID) or all observations, in addition to several others. These interfaces are explained in the sections below. The main interface used in this data group is the one for registering observations. This is when we report observations to the CEI server. With this interface we look at some of the more technical issues. Not to forget the rest of the interfaces, they are mentioned and described at a high level.

### Registration

The target url for observations is `https://sinilab.net/obs/register/`. The only part that changed is from "pos" to "obs". To be able to publish valid observation data, we send minimum the *latitude*, *longitude* and *text*. Latitude and longitude is the same as before, and "text" is a textual description of the observation. Another representation of an observation, including the previous data, is *image*. "Image" is a photo of the observation the user want to report. So an observation has two different structures that CEI accepts.

As with positions, CEI accepts requests built in a JSON format, and listing B.3 illustrates a definition of a JSON object for an observation with position (latitude and longitude) and text only. Listing B.4 shows the definition of a JSON observation object also including an image. Timestamps are omitted from observations, due to restrictions in CEI. If something goes wrong with the transmission of the observation, the server responds with a HTTP status codes in table B.1. If the observation is accepted, the server responds with a status code listed in table B.2.

```
var obsData = {
  'lat': latitude,
  'lon': longitude,
  'text': text
};
```

Listing B.3: The Definition of an Observation JSON Object

```
var obsData = {
  'lat': latitude,
  'lon': longitude,
  'text': text,
  'image': image
};
```

Listing B.4: The Definition of an Observation JSON Object with Image

An example of data being sent with the observation request, is illustrated in listing B.5. The example represents an observation without an image, including only text, latitude and longitude. The latitude and longitude again point to a position in Oslo, Norway. "Text" contains a description of the discovered incident.

```
{"lat":59.9399981075,"lon":10.6499922517,"text":"There is something fishy going on here."}
```

Listing B.5: Example of an Observation JSON Object

### Other Interfaces

CEI also provides extra functionality in terms of observations. They can be found in table B.4. However, they are only briefly described, since they are beyond the scope of this thesis.

| **Interface** | **HTTP Method** | **Description** |
|---|---|---|
| AddTag (/ID) | POST | An interface that allows the user to register new tag(s) on an observation ID. |
| ID | GET | An interface that gets a single observation identified by ID. |
| All | GET | An interface that obtains all observations. |
| AddComment (/ID) | POST | An interface that lets user register new comment to observation given by ID. |
| User (/Username) | GET | An interface that fetches data about a single user given by username. |
| User | GET | An interface that retrieves data about users in the logged-in user's group. |
| Missions | GET | An interface that gets data about missions available to the logged-in user. |

Table B.4: Other Interfaces for Observations

# Appendix C

# Answers to the GUI Test Survey

This appendix contains the "raw data" from the answers of the test survey. It should be noted that the answers are translated from Norwegian into English, due to the survey being conducted in Norwegian.

**Candidate 1:**

1. Which device did you test on?

   iPhone

2. How easy was it to understand what kind of functionality the app offers?

   Very easy

3. How easy was it to figure out what you needed to do to configure/get started with using the application?

   Easy (The app was already configured)

4. How easy was it to use the app?

   Relatively easy

5. How easy did you think it was to navigate through the app?

   The "menu" button was hard to push. It also took a while before I saw it. Both "menu" and "login" buttons were too small and too close to each other

6. Did you get sufficient information on what happened, e.g., if an error situation occurred?

Yes

7. What did you think of the user interface?

   It needs a “back” button for returning to top of page. Unclear where “send to” button sends, and what “FFI” and “CEI” means

8. Were there any functionality that you felt was missing, something that you expected to be there?

   “Blank”

9. This is a prototype that is to be further developed. Do you have any suggestions for possible improvements?

   The observation page was a bit messy, buttons could be combined. “Settings” should not be equal to “Send Position” and “Send Observation”. Should distinguish between functionality and configuration

10. Other remarks:

    “Blank”

**Candidate 2:**

1. Which device did you test on?

   Google Nexus 7, tablet

2. How easy was it to understand what kind of functionality the app offers?

   Easy

3. How easy was it to figure out what you needed to do to configure/get started with using the application?

   Easy, I think. But then again, the configuration was already done. If not, I think I would have struggled with “Settings”.

4. How easy was it to use the app?

   Relatively easy. I had no difficulties using it.

5. How easy did you think it was to navigate through the app?

   Pretty easy, but I like to have a bit more navigation buttons than just one “back” (“menu”) button. Would have liked a “home” button, and also a “back” button inside the app. Found the “menu” button, should be more visible and maybe called “home” or something instead.

6. Did you get sufficient information on what happened, e.g., if an error situation occurred?

   Yes, I felt the information was very good.

7. What did you think of the user interface?

   Things slide a bit into each other visually. Maybe use some more clear lines. Don’t quite like these alternative submissions, a bit chaotic, but don’t have any better suggestion.

8. Were there any functionality that you felt was missing, something that you expected to be there?

   “Blank”

9. This is a prototype that is to be further developed. Do you have any suggestions for possible improvements?

More navigation buttons. Results in the “Observation” part appears very far down on the page, could consider using a pop-up window only instead. In addition, this field is a bit unclear, may seem like the user is supposed to write in it.

10. Other remarks:

    “Blank”

**Candidate 3:**

1. Which device did you test on?

   Samsung Galaxy Tab and iPhone

2. How easy was it to understand what kind of functionality the app offers?

   Easy, when the menu was intuitive

3. How easy was it to figure out what you needed to do to configure/get started with using the application?

   Very little that needed configuration

4. How easy was it to use the app?

   Complicated with Alternative 1, 2 and 3. Should have self-explanatory names, and the explanation should tell what is required (not numbers, as 1+2+3). Alternatively, automatically send correct alternative according to what is filled in

5. How easy did you think it was to navigate through the app?

   Missed "back" button (back to the menu). "Login" could be a button in the main menu

6. Did you get sufficient information on what happened, e.g., if an error situation occurred?

   When the network was down, positive with pop-ups explaining what was going on. However, it felt as there was double information due to "Results" field and pop-up boxes. The "result" field may well be removed

7. What did you think of the user interface?

   Seems like "Report Position" and "Report Observation" have different functionality, but they both use "View Location" and "Get Location", so they seem very similar. Could be combined? A bit messy way of choosing image - use combo box?

8. Were there any functionality that you felt was missing, something that you expected to be there?

   Video/audio in addition to photos? If network down before you enter app, get a message about this when you open app and then just exit (since there isn't much you can do without a network connection)

9. This is a prototype that is to be further developed. Do you have any suggestions for possible improvements?

   See above.

10. Other remarks:

    “Blank”