International Master’s Thesis

Including Android Devices in PEIS-Ecologies

Peter Ericsson
Technology
Including Android Devices in PEIS-Ecologies
Studies from the Department of Technology at Örebro University

Peter Ericsson

Including Android Devices in PEIS-Ecologies

Supervisors: Federico Pecora
            Alessandro Saffiotti

Examiners: Ivan Kalaykov
           Jonas Ullberg
This thesis describes the work to make a portable ubiquitous device into a component in a networked robotic environment. The device can both control and be controlled in a robotic network as well as used to survey a robotic network. This is done by building upon existing software for making networked robotic components and use it on devices with the Android operating system. The software is the PEIS-kernel (Physically Embedded Intelligent System) middleware. The middleware makes it possible to build robotic components called PEIS-components that can communicate, work together and share their resources and capabilities. The result of this work is three-fold, first a way to make devices with the Android operating system into PEIS-devices so that they can be used to control or be controlled as well as survey a PEIS-network. With this the Android device functionality can be used to extend the capabilities of an entire PEIS-network. Secondly, devices with Android get access to a simple and powerful way to exchange information with any device that is a PEIS-device. The third contribution is the capability to make humans a much more integral part of a PEIS-network as devices with Android are in general easy to use and familiar to humans. These devices will serve as a very good human - PEIS-network interaction tool.
I would like to thank my supervisors Federico Pecora and Alessandro Saffiotti for their much appreciated guidance and help. I also want to extend my gratitude to Mathias Broxvall for his help during the development.

Peter Ericsson September 2011, Örebro
Contents

1 Introduction ................................................. 1
  1.1 Background ............................................ 1
  1.2 Objectives ............................................ 2
    1.2.1 Motivation ...................................... 2
    1.2.2 Primary goal .................................... 2
    1.2.3 Secondary goal .................................. 3
    1.2.4 Target users/platforms .......................... 4
  1.3 Outline ............................................... 4

2 Background .................................................... 7
  2.1 Related work in network robot systems ................. 7
  2.2 Portable interaction devices in network robot systems . 8
  2.3 Related work in software for network robot systems . . 8
  2.4 Specific background on network robot systems ......... 9
    2.4.1 Idea .......................................... 9
    2.4.2 Workings and functionality .................... 9
    2.4.3 Existing PEIS-middleware ports ................. 14
    2.4.4 Missing piece for PEIS-ecologies ............... 15
  2.5 Specific background on interaction tool with Android . 16
  2.6 Android system ....................................... 17
    2.6.1 System construction ............................ 17
    2.6.2 Available versions ............................. 18
  2.7 Limitations ............................................ 19
    2.7.1 JNI not JNA ................................... 19
    2.7.2 Mobile devices ................................ 19
  2.8 Eclipse ............................................... 19
    2.8.1 Android development in Eclipse ............... 19
  2.9 Feasibility analysis .................................. 20
    2.9.1 Summary ....................................... 21
List of Figures

1.1 The layered design of the project, based on the project objectives 4
2.1 Example of co-operation in a PEIS-ecology .......................... 10
2.2 The RFID-tags under the floor in PEIS-home1 ...................... 12
2.3 The Giraff robot are build to be used in homes ...................... 12
2.4 The Nao robot can be integrated into a PEIS-network ............ 13
2.5 The Roomba vacuum cleaner from the company iRobot .......... 13
2.6 The kitchen of PEIS-home1 ........................................... 14
2.7 The living room in PEIS-home1 ...................................... 14
2.8 The bedroom in PEIS-home1 ......................................... 14
2.9 A mote device from PEIS-home1 ..................................... 15
2.10 The Android system .................................................. 18
2.11 Emulation of mobile phone ........................................... 20
3.1 The layers of code in PEIS-JavaJNI and PEIS-Android .......... 26
3.2 Classes in the java object layer ....................................... 30
3.3 Sequence diagram that shows what happens at a subscription .. 31
3.4 Sequence diagram that shows how a tuple is published .......... 32
3.5 Sequence diagram that shows what happens at a tuple retrieval 33
3.6 Interface of the example application getTupple ................. 34
4.1 Plot of execution time to subscribe to tuples .................... 40
4.2 Plot of execution time to publish tuples ......................... 41
4.3 Plot of execution time to retrieve tuples ......................... 42
List of Tables

4.1 Overall results for developed Java packages and test application 36
4.2 Overall metrics results for developed Java code .................. 37
4.3 Metrics values for the developed classes in PEIS-JavaJNI ....... 37
4.4 Metrics values for the classes in PeisAndroid and getTuple .... 38
4.5 Execution time in milliseconds for subscribing to tuples ....... 40
4.6 Execution time in milliseconds for publishing tuples .......... 41
4.7 Execution time in milliseconds for retrieving tuples .......... 42
4.8 Metrics values for the classes in the developed PEIS-JavaJNI .. 45
Chapter 1
Introduction

The field of networked ubiquitous robotic devices is growing rapidly as more and more device types are capable of exchanging information. A robotic device is in this work seen as any system with capabilities as to measure, control and communicate with the robotic ecology and environment in which it is deployed. This can be a simple sensor node that only measures a single physical value such as temperature. The device can also have the ability to control the environment and ecology and use the capabilities and resources of other devices in the ecology. An example of this is [31], where robotic devices work together and share information and in doing so accomplish a result that is higher than the sum of the individual device capabilities.

The result of this work is an extension and adaption of an existing middleware for design of network robot systems. The result makes it possible for portable devices such as mobile phones to be used as components in network robot systems.

1.1 Background

The general notion of a robot is a device that is capable of many things without much help. Today there are a number of robots as the Asimo from Honda [6] and the Nao from Aldebaran [30] that can accomplish a great many things on their own. These robots are so called humanoid robots that mimic the way a human moves, they have two legs and two arms with grippers (fingers) at the ends. A problem with the idea of a single device as a robot that do all tasks is that the device needs to be quite complex to accomplish all possible tasks. A way to get around this problem, is to not concentrate all the needed functionality in a single device but to distribute the functionality and resources to devices in the environment. In this way a robotic ecology is created. A task would then be accomplished through cooperation of devices with some functionality and the addition of a device, may it be a weak one, would add to the collective capability of the robotic ecology. The ecology would in this way grow so that
even the weakest link in a sense would have the capability of the entire ecology through interaction and collaboration. This work builds upon the **PEIS-ecology** study, explained in [Section 2.4], that was done at **Applied Autonomous Sensor Systems** (AASS) at Örebro university [1]. PEIS stands for **Physically Embedded Intelligent System**. A PEIS-ecology contains PEIS-components that run on PEIS-devices, the components can communicate with other PEIS-components in some way and have some functionality that is useful for the PEIS-ecology or those who use the ecology. All the PEIS-components are resources in the PEIS-ecology and both their **functionality** and **data** is accessible in the PEIS-ecology for other PEIS-components through a distributed database. The functionality and data of PEIS-components are published in this database so that other PEIS-components can use them and get aware of their existence. The data and functionality is distributed in the database with so called **tuples**, which are packets with data or commands. The PEIS-components react to the changes in the database and that gives intelligent behavior which is the objective for a PEIS-ecology.

### 1.2 Objectives

#### 1.2.1 Motivation

The combination of networked robotics and modern mobile ubiquitous devices such as modern mobile phones is a powerful mixture of functionality and capabilities. With the inclusion of new device in a PEIS-ecology that humans are both familiar to and comfortable with, humans can be made a much more integral part of PEIS-ecologies. It gives humans an easy way to control and survey a PEIS-ecology, as well as letting the devices in the PEIS-ecology have a more direct connection to humans that use the PEIS-ecology. A modern portable device with the **Android operating system** such as a modern mobile phone is a good choice for a human - PEIS-ecology interaction tool. Because most modern portable devices are easy to handle and they also have a rich set of functionality such as quite big and good screens, sensors and wireless interfaces which can be made a part of the PEIS-ecology. The Android operating system is a good choice as it is used on many different devices that could gain from being a part of a PEIS-ecology. Android is also a more open system than other portable device operating systems as a result of that it is developed in a open source project [3]. So in conclusion PEIS-ecologies would get a very capable new component type and the devices with the Android operating system would get access to all the possibilities of PEIS-ecologies.

#### 1.2.2 Primary goal

Below are the four primary objectives for this work described. Each of the four objectives for this work have a part to play in the overall design and figure 1.1
shows how each objective builds upon the previous objective. The main users in mind are those that are familiar with the PEIS idea, the PEIS-component developers that use the programming language Java for computers and Android application developers that construct PEIS-components.

**Android device as a PEIS-ecology device** The most important objective is to be able to run PEIS-components on a portable device that uses Android as the operating system. This objective ensures that devices with Android can be made a part of a PEIS-ecology, if they can communicate with the rest of the PEIS-ecology in some way. This objective can be accomplished by either recreating the needed functionality for communication in a PEIS-network or by using existing software for constructing PEIS-components. This objective corresponds to layer 1 in figure 1.1.

**Ease the creation of PEIS-components in Java** The second objective is to make it easier to create PEIS-components in the programming language Java. This is done so that the result will be easier to use in different Java projects, large changes should not be needed when switching PEIS-component project. This objective is layer 2 in figure 1.1.

**Object oriented PEIS-component programming** The third objective is to further ease the process of creating a PEIS-component by adding object orientation to the design and divide the functionality to different objects. By the addition of this layer it will be clearer for the PEIS-component developer how and what can be done in a certain context. Up until this objective (layer 3 in figure 1.1) the target user is both computer and Android software developers that make PEIS-components. This is also the first layer that is intended to be used by developers.

**Android aware PEIS-component** The fourth objective is to build upon layer 3 by providing the structure for easy access to Android specific functionality in layer 4, which is the highest layer in figure 1.1 before the PEIS-components. This layer will take advantage of both the underlying layer 3 and the Android device specific functionality as access to hardware. This objective will make it easier to develop PEIS-components that use functionality specific to Android systems. This is the second layer that is intended to be used by developers.

### 1.2.3 Secondary goal

The main target users of this work are Java and Android application developers that develop PEIS-components. The secondary goal of this work is to provide a way for the Java and Android application developers that want to use the functionality of PEIS-ecologies but are unfamiliar with them. This broadens the user
base as more people would find the inclusion of the PEIS-ecology functionality easily attainable. The demands this place on the development are a clear structure for usage and also a good documentation.

1.2.4 Target users/platforms

This work targets Java programmers and Android application developers that want to include the functionality of PEIS-ecologies into their design. There are no specific restrictions for the type of platform that the code would be used on. The platform only need to be able to run C-code and Java-code that use Java Native Interface \[32\]. Java Native Interface is a technique that is used to allow C-code and Java-code to interact.

1.3 Outline

The rest of this thesis is organized as follows:

Chapter 2 Explains the background to this work as specific previous work, motivation as to why this work is useful, how it can be used. Also limitations connected to development and usage will be discussed.

Chapter 3 Describes the design patterns that were used and the resulting design of the work.

Chapter 4 Contains an evaluation of the resulting software to determine if the objectives have been fulfilled and to what degree of quality.
Chapter 5 Contains the end conclusion from the results and the evaluation. This chapter also takes up possible future work such as how this work can be extended and refined.
Chapter 2
Background

This chapter describes the foundation to this work and also related work in the field of network robot systems, intelligent spaces, ubiquitous robotic spaces and mobile devices as control interfaces. Knowledge in these fields as well as in C- and Java-programming was needed for this work. However this thesis is written so that any one with a little knowledge of computers should be able to understand the main concepts of this work. The two enabling technologies for this work, Android and the PEIS-software are described in more detail respectively in section 2.6 and 2.4 as they are crucial to this work. This chapter also take up limitations for this work and the tools and resources that were used. All tools that were used for this work is freely available without any cost, these are the Android development tools, the programming development tool Eclipse and the software for building PEIS-components. Even though the Android operating system was not used for development per-se it should be pointed out that it also is freely available without any cost, and it is developed in a so called open source project.

2.1 Related work in network robot systems

There are a great many number of publications to gain insight from for a project like this. My work builds upon the PEIS-ecology study [35] done at Örebro university. The idea of this work is to abstract away the notion of a robot as a single entity. A PEIS-device can be any device with some way to communicate or be used in a PEIS-network. From a simple RFID-tag to a device that can move around the environment and accomplish a more complex set of tasks. Another central point in the PEIS-ecology idea is that the PEIS-devices can share their resources and capabilities among the PEIS-devices in a PEIS-network.

Other work that is related to this work takes up the idea of ubiquitous robotic spaces [37] in which a mobile robot works in an environment with sensors to survey an area. The research in intelligent space [19] takes up the idea of a
system that understand what is happening in a space with the help of a set of
sensors. In the study of an artificial ecosystems [36] the idea of having a mul-
tiagent, distributed system for mobile robots is investigated in an environment
with distributed sensors and actuators where the devices co-operate.

The choice to use a PEIS-system for this work is motivated by a few important
facts. The PEIS-ecology study is currently (September, 2011) the largest study
in the field of network robot systems with devices that work as a collection of
co-operating devices where the work has resulted in a mature code-base. The
code is well documented and another very important factor is that it is used
ongoing and tested in three different test environments where one is outside the
university in a apartment that has been equipped with various test equipment
for a PEIS-environment/smart-home in a collaboration between the university
and various companies.

2.2 Portable interaction devices in network robot
systems

The human - machine interface to a network robot system is important because
a good interface makes it easier to integrate humans in network robot systems
and for humans to control and survey such a system. In [29] a system is de-
scribed that automatically creates an interface based on what is possible and
displays it on a mobile device, in [28] a framework to automatically generate
interfaces for smart phones from abstract specifications is presented. In [25] a
mobile learning system is presented where a mobile phone with Android is used
as a mobile laboratory. The system is used to show measurements of power
supplies that are used in the laboratory, the users can also use the system to
comment the measurements and place questions to participants or the teacher.

2.3 Related work in software for network robot
systems

Jayedur Rashid [18] describes the work to include tiny devices and humans in a
network robot system in his doctoral thesis [34]. The work results in a new de-
vice type, the tiny device, that brings many new possibilities to PEIS-networks.
The tiny devices use a port of PEIS-software for tinyOS [38] (an operating sys-
tem for small micro controllers). The tiny devices can be a small sensor and
actuator devices that now can be made a part of a PEIS-network. The commu-
ication between the tiny devices tiny network and a PEIS-network go through
a gateway device, the Tiny-gateway. The Tiny-gateway translates between the
minimalistic communication for the tiny devices and the regular PEIS-network
communication standard. From the PEIS-network the tiny devices are perceived
and used as any other PEIS-devices. The special communication for the tiny devices is used so that devices with limited memory and communication resources can be used.

2.4 Specific background on network robot systems

As this work builds upon the **PEIS-ecology study** the PEIS-ecology will be explained more detailed in this section.

2.4.1 Idea

The main idea of a **PEIS-ecology** (**Physically Embedded Intelligent System**) is that robots are abstracted away and what is left is a **PEIS-network** with **PEIS-components** that publish and subscribe to resources and functionality of other PEIS-components. A PEIS-network consists of PEIS-components that run on **PEIS-devices**. A PEIS-device is a component of the PEIS-network that has some kind of resource or functionality and can communicate in some way with the rest of the PEIS-network. Multiple PEIS-components can run on a single PEIS-device such as a computer. All resources and functionality in a PEIS-network are accessible through a distributed database called the **tuplespace**.

The carrier/package for data, commands and functionality in the PEIS-network is called a **tuple**. A PEIS-component gets tuples from a PEIS-network by **subscribing** to tuples and distributes tuples on the PEIS-network by **publishing** tuples. Figure 2.1 show an example of a mobile robotic table that asks a fridge to give it a can of soda. This is made possible with tuples. The fridge publishes tuples that hold an inventory to its content and what functionality it can provide. In this case the fridge has the functionality to open and close its door and reach out an item that it contains. The robotic table can read the tuples from the fridge and determines if the wanted can of soda is available, the robotic table then go to the fridge and ask it to open its door and give it a can of soda. The table can follow the execution of the commands as the fridge status is also published as tuples. When the door is fully opened the table can move closer to the fridge so that the can of soda can be safely placed on top of the robotic table. The table then moves away and tells the fridge to close its door.

2.4.2 Workings and functionality

All PEIS-components run the **PEIS-kernel** that is written in the programming language C. The PEIS-kernel is part of the **PEIS-middleware** that provides additional functionality and ease the use of the PEIS-kernel functionality. The PEIS-kernel takes care of communication between PEIS-components so that the tuplespace is updated and tuples with data and commands are distributed
Figure 2.1: Example of a robotic table co-operating with a robotic fridge in a PEIS-ecology.

in the PEIS-network. The actual PEIS-network can use TCP or UDP to communicate but primarily TCP is used. There are several layers of data and message handling in the PEIS-kernel software. At the top, the highest abstraction level from the raw data communication, the data is handled as so called tuples that act as data and information vessels, this is what the PEIS-component developer (the programmer) use. Below this layer is a peer-to-peer network (P2P network) between the PEIS-devices in the PEIS-network. At the lowest layer is raw data handling with TCP and UDP sockets. As the PEIS-kernel handles the network and communication between the PEIS-components the user does not need to worry about handling disappearing and reappearing PEIS-components. And this is one of the strengths of the PEIS-network design that it can work in a highly dynamic environment.

The data and information in a PEIS-network is carried around with tuples, these do not only carry data and information but also have the task of controlling the components and the network. The tuples can be seen as entries in a database that is distributed over all the components in the network. Each PEIS-component can publish and subscribe to tuples. A subscription to a tuple works much like a newspaper subscription. The subscription gives access to information, in this case a tuple. When a subscription for a tuple is made, that tuple will be made accessible for the PEIS-component if it exist. As a tuple is published all components in the network that subscribes to that tuple will automatically get updates for it. Each tuple is identified with up to a seven (7) level name such as houseA.roomB.cornerC.levelD.*.rawData.rgb where the character * works as a wild-card, meaning that the wild-card position could be anything. A subscription with a wild-card in the name at a level results in a subscription to
all possible tuple name-levels for that position. The second type of information used to identify a tuple is the owner of the tuple, that is an integer number. A component may very well publish a tuple on the network with the same name as a tuple from another component, but the owner for the tuple that is published will be different. There exists a wildcard in the form of -1 for owner so one could get, as an example, all camera data no matter who published it.

There exist several tuple-types that can be used in a PEIS-component by a programmer:

**Normal tuple:** Used for distributing data and information in the PEIS-network.

**Meta tuple:** Can be used to set up to and from which tuples the PEIS-components reads and writes data, they work as pointers to normal tuples.

**Abstract tuple:** Used for subscriptions, the abstract tuple represent a tuple that is wanted, it is a template that can contains so called wild-card entries for fields as name and owner.

The PEIS-kernel is currently at *generation 6*, that is the stable version. During this work the PEIS-kernel version 4, 5 and finally 6 have been used. The switch to generation 5 of the PEIS-kernel code was made because it is more efficient and is also designed to work both on **32-bit** and **64-bit** systems, which was the main reason for the switch to that version (after some quite time consuming debugging). As generation 4 of the PEIS-kernel does not take into account differences for 32-bit and 64-bit systems the data sent from a system could in generation 4 have different sizes depending on if the PEIS-component was on a 32-bit system or a 64-bit system. Generation 6 of the PEIS-kernel is more efficient and has a more well defined user interface than the previous versions. This work now uses generation 6 of the PEIS-kernel to gain from the improvements. The different versions of the PEIS-kernel are incompatible in PEIS-networks, this means that all PEIS-components in a PEIS-network need to use the same version of the PEIS-kernel. Otherwise the tuples that are sent between different PEIS-components with different PEIS-kernel version will not be read in a correct way.

The PEIS-middleware that contains the PEIS-kernel and the PEIS-ecology idea is continuously used and updated at Örebro university. Three (3) test environments for PEIS-components exist and two (2) of them are on the university campus. The two test environments mimic two ordinary apartments and are equipped with cameras for positioning, recognition and volumetric estimation, RFID-tags under the floor (see figure 2.2 that show how the RFID-tag were installed) for positioning, sensors for environment information, actuators for environment control and computers that can be used as PEIS-devices. The third test environment for PEIS-ecologies is located in a real apartment that is
CHAPTER 2. BACKGROUND

managed in a collaboration between the university and several companies. The apartment is used for more real world tests of technical solutions that are mature enough to be used in a live environment. One example of such devices is the Giraff robot (figure 2.3, developed at AASS [1] in collaboration with Giraff Technologies [11]) that can be used to look after and keep in touch with people when time and distance otherwise would make it difficult.

Figure 2.2: The RFID-tags was installed under the floor in PEIS-home1.

![Image 1](image1.png)

Figure 2.3: This is an example of a device that is technologically mature enough to be used in the third test environment for PEIS-ecologies.

There exists several different mobile robots that can work as a part of the PEIS-networks in the two test environments. Examples of these robots are the Nao from Aldebaran [30] that can be seen in figure 2.4 and a robotic vacuum cleaner called Roomba, from the company iRobot [16], that can be seen in figure 2.5. These robots are devices that would benefit from a human-machine interface in the form of a smart phone. Both the control of robotic devices and the presentation of sensor readings from these devices could be handled with a smart phone. Control and presentation of data from the devices could be done through the smart phones touch screen as an example. TEST TEST.
2.4. SPECIFIC BACKGROUND ON NETWORK ROBOT SYSTEMS

The first built test environment for PEIS-environments is called PEIS-home1. Figure 2.6 shows the kitchen in the so called PEIS-home1, figure 2.7 shows the living room and figure 2.8 shows the bedroom. PEIS-home1 is built like a bachelor apartment that is quite compact. The second test environment on the university is called PEIS-home2, it is bigger and has a different set of hardware in it such as more RFID-tags under the floor for more accurate positioning. The third test environment that is a normal apartment outside the university has equipment installed from both the university (PEIS-network and RFID-tags under the floor) and some companies. This test environment is quite new (September, 2011) and is used for more real tests and to show developed mature solutions from the university and companies. It do not contain many PEIS-devices at this time, but that will most likely change during the coming months.

PEIS-ecologies can be very heterogeneous systems with a large number of different devices such as motes, different robots and computers. A mote as in figure 2.9 is a device that usually has a small CPU and is connected to the PEIS-network. It is used for sensing and actuation in a PEIS-ecology. As a PEIS-ecology can be very heterogeneous the PEIS-middleware has been adapted to work in several programming environments, called ports of the PEIS-middleware.
So that the PEIS–middleware can be used for many different programming languages and for many different hardware devices.

2.4.3 Existing PEIS-middleware ports

Currently (September, 2011) there exist several ports and implementations of the PEIS-kernel/PEIS-middleware.

PEIS-Kernel: The original version of the Kernel currently used on all computers and Robots.

Tiny PEIS-Kernel: A TinyOS [38] port of the PEIS-kernel suitable for Motes and other tiny devices. Developed by Jayedur Rashid [18] in his doctoral thesis work [34].
2.4. SPECIFIC BACKGROUND ON NETWORK ROBOT SYSTEMS

Figure 2.9: A so called mote (sensing, actuation and computing device) from PEIS-home1.

PEIS-Lisp: A native Lisp interface to the PEIS-kernel. LISP is a programming language that is well known and common in computer science. The reason for this is that it is very often used when developing artificial intelligence programs. A PEIS-middleware port that makes it possible to use with LISP is quite useful as some kind of artificial intelligence is most likely part of a PEIS-ecology.

PEIS-JavaJNA: A native Java interface to the PEIS-kernel. For development of PEIS-components in Java the PEIS-JavaJNA library can be used. PEIS-JavaJNA works by accessing the native C-code of the PEIS-kernel and supports development of multi-threaded applications. PEIS-JavaJNA uses JNA (Java Native Access) [17] to access the underlying native C-code in the PEIS-kernel. The Dalvik virtual machine used in Android does not support the easier way to access C-code from Java with JNA so JNI (Java Native Interface) [21] needs to be used. PEIS-JavaJNA is compared to the results from the work described in this thesis [Chapter 4], called PEIS-JavaJNI that use JNI (Java Native Interface).

PEIS-Lua: A Lua interface to the PEIS Kernel. Lua [22] is a fast, lightweight and robust scripting language. Lua takes up a small footprint and can be run on all varieties of UNIX and Windows, it also runs on many mobile devices and on microprocessors such as the ARM [5] and Rabbit [33]. Lua can both easily extend and be extended with code written in other languages. The Lua port of the PEIS-middleware gives access to the power of Lua in that it is fast, lightweight and robust.

2.4.4 Missing piece for PEIS-ecologies

Even with all the different ports and implementations of the PEIS-kernel and the PEIS-middleware there is a missing piece for PEIS-ecologies. An easily portable
device for control and surveying of a PEIS-network that can make humans into a more integral part of PEIS-networks. Even a small portable computer can be cumbersome at times, but a modern mobile device such as a smart phone is a good choice. Most smart phones are easy for humans to keep with them and to use. They also give the added value of having a large set of sensors, wireless interfaces and many of them also have big and good screens. To have a smart phone as a PEIS-device gives humans an easy way to control and survey PEIS-networks that is always at hand. A smart phone also includes humans in PEIS-networks in a much better way, thanks to that the device is easy to have at hand, easy to use and the wireless interfaces allows humans to move around in the environment that house the PEIS-ecology. So in conclusion a good portable device as a smart phone could work as a good human-robot interface for network robot systems. For the reason that smart phones are familiar and easy to use for humans and that would also expand the user base for PEIS-ecologies.

2.5 Specific background on interaction tool with Android

The use of JNI as the means to access C-code on an Android device has been tested to shorten the development time for projects that access C-code from Java code. This is important for this work as the PEIS-functionality is available in C-code and Android applications are written in Java code. In [23] the work to make a web application engine work on the Android platform is described. The web application engine is called xFace and it is written in C/C++ so JNI is used to be able to use it on the Android platform. The result of the work is that reuse of the xFace web engine shortens the development time and cost. But because JNI is used the efficiency is negatively effected as a result of the need to make calls from the C/C++ side to the Java side. The authors point out that this is most notable with applications that make many calls from the C/C++ side to the Java side as with highly graphical games that need many calls to draw on the screen.

The value of using smart phones with Android as a human-machine interface, as in this work, has been proven to be able to give a systems that is easy and cheap to use. The added value of using a smart phone is that it most often got a large variety of sensors and a big screens that can be used in an application. In [13] a so called gymkhanas game is implemented, it is a game type where tasks are solved one after another as the participants visit a series of places. The work uses mobile phones with Android to give the game participants tasks and questions. The participants also use the mobile phones to give their response to the tasks and questions. The result of the work is a system that is easy and cheap to use. The solution to use mobile phones with Android also give the
added value that the mobile phones positioning and camera capabilities can be used as a part of the game. In [14] a method to create adaptive user interfaces, for a mobile phone with the Android platform, that change according to the situation and context is presented. The method uses machine learning techniques to change the user interface. The method uses sensor readings, weather data, time and previous choices as input. The interface response is from the start set to supply a response to some situations but learns the choices of the user to provide a better interface. The average accuracy to predict the right interface for the user was on average 69 percent.

In [10] the iFall application is presented. It is an application developed for Android devices that detects when a human falls. The application is aimed for the elderly and the aim is to minimize the time a human needs to lie on the ground/floor after a fall. The application uses the accelerometer of the Android device to detect when the acceleration gets above a defined threshold. When a fall is detected, pre-defined contacts are contacted and supplied with location information and time of the fall. If a contact responds to the message a bidirectional communication line for voice communication is set up. Emergency services are contacted when a contact confirms the fall or if no contact responds to the distress message. An application like the iFall can be used in a PEIS-environment/smart-home, the PEIS-environment can help in case of accident by alerting humans close by and also send robots that can help.

2.6 Android system

Android [2] is an operating system built upon a Linux kernel, it also include a software stack and a set of applications. Android is developed as an open source project in the so called Android Open Source Project (AOSP) [3] led by Google. Most of Android is released under the Apache Software License 2.0 [4], other parts as the Linux kernel patches in Android are released under the GNU General Public License version 2 (GPLv2) [12]. Any one can use the code and make whatever changes they want as device manufacturers that can make changes to the code to customize it for their devices.

2.6.1 System construction

An Android system is built up as figure 2.10 show. At the bottom layer is a Linux kernel with version 2.6 (September, 2011), that handles the access to the underlying hardware and acts as an abstraction layer between the hardware and the higher levels of the software stack. Over the Linux kernel is a set of C/C++ libraries, the functionality of these libraries are available to the application developer through the application framework. The run-time includes core libraries that give the run-time most of the Java language functionality as Android applications are written in the Java programming language. The used
Figure 2.10: The Android system (operating system, software stack and applications).

The virtual machine is the Dalvik virtual machine, it is optimized for memory and execution efficiency. The Dalvik virtual machine does not execute Java byte code as ordinary Java programs are compiled into, the programs are compiled into the Dalvik Executable format (DEX-files) which is optimized for memory efficiency. The next level of the system is the application framework which provides functionality for writing the applications. The last level contains the Android applications that are shipped with the system. These are for example programs for SMS and e-mail messages and a web-browser.

### 2.6.2 Available versions

The current versions (*September, 2011*) of the Android system are version 2.3.3 (called Gingerbread) for mobile devices as mobile phones and version 3.2 (Called Honeycomb) that is for larger mobile devices as tablets. A coming version (called Ice Cream Sandwich) is said to combine the Gingerbread and Honeycomb versions into one version for both mobile phones and tablets.
2.7 Limitations

2.7.1 JNI not JNA

A PEIS-component developed for the Android system needs to be able to use the underlying C-code in the PEIS-kernel. The Dalvik virtual machine does not however support the easier way to access C-code from Java with JNA. This means that the more complicated and time consuming technique with JNI has to be used. By using JNA, only Java code needs to be written as opposed to JNI (Java Native Interface) [21] which requires interface code between the C and Java code. The extra layer the so called glue-code or interface code between the Java-side and the C-side result in more work as each native function (C-function or more precise a JNI wrapper to the C-code function) that need to be accessed must be declared at the Java-side. The interface code translates between the Java-side and C-side, for example Java Strings are translated into char-types.

2.7.2 Mobile devices

Android is the base for many portable devices and the majority of them are phones with a screen size of up to about 4 inches, unless a so called tablet is used which can have a much larger screen. The screen size poses some limitations as it can be hard to fit everything needed into a single screen that is 4 inches and an application should be able to work for both the smaller mobile screens and the larger tablet screens. The device used for testing in this work is the HTC Desire HD [15] which has a 4.3 inch screen with a resolution of 480 times 800 pixels, a CPU that runs at 1 GHz and a 8 megapixel camera.

2.8 Eclipse

The main development tool that was used for this work is Eclipse [9]. Eclipse is an IDE (Integrated Development Environment) which means that it includes a set of development tools, in this case a code editor, a compiler and a debugger. Most part of Eclipse is written in Java. It has the ability to use plug-ins to add functionality for development in many languages such as C/C++, Python, Ada and Perl. The plug-ins can also as an example also add functionality for code analysis, documentation and additional debug tools.

2.8.1 Android development in Eclipse

Google provides a Software Development Kit (SDK) for development of Android applications which includes tools to debug, compile and install Android applications on a mobile phone with Android. Google also supplies a very good plug-in for Eclipse for development of Android applications, that is called ADT
which stand for Android Development Tools. The ADT brings the power of the SDK into Eclipse so debugging, compilation and installation can be done from Eclipse. One very useful part of the SDK/ADT is the possibility to test applications in an emulated mobile phone. The emulator can be adjusted in many ways to emulate many different devices and settings. As an example different physical devices can be set as well as Android version, memory size and much more, figure 2.11 shows an example of an emulation of a mobile phone running Android version 2.2. To make it easier to develop Android applications that use C-/C++-code, Google has made the Native Development Kit (NDK) which makes it possible to compile C-/C++-code for an Android device and use the code in an Android application. The NDK also gives access to a set of native code that will be supported in all future versions for Android from version 1.5.

2.9 Feasibility analysis

From the available tools as the SDK and the ADT from Google and the capabilities for the Android platform some conclusions can be drawn for this project. The first objective of running a PEIS-component on an Android device is possible. Either by using the existing PEIS-kernel C-code and access it with JNI or recreate the functionality of the PEIS-kernel in Java code. To recreate the needed functionality in Java code would demand a great deal of time and work, and there is an added value of reusing the C-code PEIS-kernel code. In [20] the efficiency of C-code on Android device is compared to the Java-code counterpart. The results show that large performance gains is possible for memory and
calculation operations if they are performed in C-code libraries instead of in Java-code. The authors show that the JNI communication give a slight delay in comparison to the Java operation counterpart, but it is not a factor when set in relation to the memory and calculation operation times. The conclusion is to use the existing PEIS-kernel C-code together with JNI on an Android device. The evaluation of this work [Section 4.2] shows that the performance for PEIS-JavaJNI is good when compared to PEIS-JavaJNA.

The second objective to ease the use of the developed code for developers can be fulfilled by encapsulating the access to the PEIS-kernel C-code via the JNI code in Java-code.

Objective three to add an object oriented user layer to the developed code is a matter of using the code from the lower objectives and present an object oriented interface for the end user, the developer of PEIS-components in Java and Android application developers.

Objective four is as objective three a matter of using existing Java-code for the PEIS functionality and the Android specific functionality and present a good interface for the end user, the Android application developer of PEIS-components.

As for the secondary goal to give Java and Android developers that are unfamiliar with PEIS-ecologies an easy way to use the functionality of PEIS-ecologies, it is possible by designing the software in a clear and understandable way with a good documentation. A getting-started guide can also be created as a complement to the documentation.

2.9.1 Summary

The two main users that this work is aimed for is the Java developers that develop PEIS-components in Java and the Android application developers that develop PEIS-components for Android. The Android platform is a powerful base for systems that PEIS-components could benefit from, and with the possibility to run PEIS-components on Android devices the Android world would gain the capability to access and use PEIS-networks. So this project is beneficial to both the PEIS-ecology and the Android side. The goal for this project is divided into four primary objectives and a secondary objective.

Primary objectives:

Objective 1: Make it possible to run a PEIS-component on an Android device.

Objective 2: Make it easy to use and access the functionality needed for a PEIS-component.
Objective 3: Add a clear and understandable interface to the functionality needed to develop a PEIS-component for both computers and Android devices.

Objective 4: Make it easier to use the benefits of an Android system in a PEIS-component.

Secondary objective: To give Java and Android developers that are unfamiliar with PEIS-ecologies an easy way to use the functionality of PEIS-ecologies.

The next chapter will show the resulting software, how the different objectives were realized, designed and implemented and how the software is intended to be used.
Chapter 3
The PEIS API for Android

This chapter will show how the resulting software, PEIS-JavaJNI, was designed and also explains the design choices. The resulting software is mainly compared to the already existing PEIS-JavaJNA, that is used to develop PEIS-components in the programming language Java. PEIS-JavaJNA uses the PEIS-kernel for the PEIS-functionality. The PEIS-kernel is written in the programming language C and PEIS-JavaJNA access this code with Java Native Access (JNA). To use C-code on an Android device JNI and not JNA is used, that is why the existing PEIS-JavaJNA is not used right away to get the PEIS-kernel functionality on an Android device.

3.1 Introduction

The first objective for this work is to be able to run a PEIS-component on an Android device. As mentioned the existing PEIS-JavaJNA use JNA to access the C-code in the PEIS-kernel. As JNI is the method used to access C-/C++-code in Android applications PEIS-JavaJNA can not be used right away for an Android application that would work as a PEIS-component. There are two choices to get the needed functionality for a PEIS-component on an Android device. Either to access the PEIS-kernel C-code from the Java-code in the Android application or to recreate the PEIS-kernel functionality in Java-code. Even though it would be possible to recreate the PEIS-kernel functionality in Java-code, there are a few factors that speak against such a design choice:

First: A reimplementation in Java-code would create another PEIS-kernel that would need updating when the original one was updated.

Second: The existing PEIS-kernel contains a large set of C-code that would have to be rewritten in Java-code and this is both time consuming and error prone because of the differences between the programming languages.
Third: Java-code is most times not as efficient as C-code when it comes to execution speed and memory requirements, which is an important factor for portable devices.

What speaks for a Java-implementation of the PEIS-kernel is that it would be more coherent as it would be written in one programming language. It would also be a bit easier to use as Java is the main programming language when making Android applications.

A solution to use the existing PEIS-kernel written in C-code and accessing it with JNI would not give multiple PEIS-kernels to maintain. This solution would not require time consuming reimplementations of the needed functionality and it would most likely be more efficient than a Java implementation. The negative part of such a design is the mix of programming languages (C and Java) and also the use of JNI that could have a negative effect on maintenance of the code as it would require additional knowledge of the programmer that would maintain the code. The more complex code structure when using Java, C and JNI-code could be hidden from the end user, Java/Android application developers, so that a Java interface would be presented. In this way the use of a solution with JNI would circumvent the problems with a reimplementations of the PEIS-kernel functionality. The negative part of a solution with JNI is that a developer that used the developed software would have to understand how to access the PEIS-kernel functionality with JNI. This problem is the main focus of objective two (2).

The second objective is to ease the access to the PEIS-kernel functionality. In a simple solution with JNI the PEIS-kernel functions needed in a Java/Android application would have to be declared in the application. Also the layer of JNI-code, the code that translates between the Java-code and C-code side would have to be changed between different applications. The reason the JNI-code needs to be changed between applications is that the functions in the JNI-code are linked through their name to the Java class that calls them, so if the name of the Java class changes the JNI-code functions names needs to be changed. A solution would be to add a layer of Java-code between the application and the JNI-code that then would stay constant between applications and take away the need to change the JNI-code.

Even though objective two would make it easier to use the functionality of the PEIS-kernel in a Java/Android application, the underlying Java-code would be a wrapper of PEIS-kernel functions. The third objective is aimed at improving the code from the second objective by adding object orientation to the design. Figure 3.1 shows the layered design that is the result of adding each objectives code to the overall design. At the bottom is either a computer or Android device that the PEIS-component executes on, over that is the PEIS-kernel and then
3.1. INTRODUCTION

code from each of the four (4) objectives. A PEIS-component can either run as a component written in Java for a computer or an Android device (component A and B) or a component that run on an Android device (component C and D). The difference between the components that can run on both a computer and an Android device (components A and B) and the Components C and D is that these use Android system specific functionality. This functionality is the main focus for objective four. The code from objective two (2) is divided to two boxes named JNI Layer C-code and JNI Layer Java-code. These two boxes are a Java-bindings layer that connect the C-code and the Java-code.

The third objective is to improve the interface to the code for the end user, the Java or Android application developer, so that it is more clear how to do certain tasks in a certain context. The way this can be done is to add another layer of Java-code that use the previous layer with easy access to the PEIS-kernel functionality. Either a better interface can be created with only a structured naming convention of the Java functions or the new layer could also use the inherent power of the Java programming language by adding object orientation to the design, moving away from the function oriented interface of the previous layer. The benefits of only using a structured naming convention would be less work in implementation and reduced complexity of the resulting code. The benefits of an object oriented interface is that it would create a clear distinction between the functionality needed in a certain context and what is possible, as the functionality is tied to certain objects. The negative part of an object oriented interface is that it takes a little more time to implement, but the benefits outweigh this negative aspect. Up until the third objective the users are both Java and Android application developers that need PEIS-functionality. In figure 3.1 the code for this objective is named Java object layer.

The fourth objective is to provide the structure for easy access to the Android device and Android platform for the PEIS-component developer. The Android platform can provide many benefits for a PEIS-component running on an Android device. Access to Android device functionality in combination with PEIS-functionality would be a very good combination as developers that are unfamiliar with the Android platform could get easy access to this functionality and it could also ease the creation of PEIS-components that use Android specific functionality. Figure 3.1 shows the layered design that is the result as each objectives code is added to the design. In figure 3.1 the code for this objective is called Android abstraction layer.

The secondary goal for this work is to ease the use of the developed software for Java and Android application developers that are unfamiliar with the PEIS idea and code. For this objective four (4) is a good start but equally important is documentation on how to use the developed software. For this, documentation
can be written into the code as comments and can be either read directly from the source files or extracted with a Doxygen tool [8]. Another addition that could help in understanding how to do tasks for a PEIS-component is an example application that shows how to do common tasks on a PEIS-component on Android. With these two additions the code should be easier for those who are new to PEIS to use the code.

### 3.2 Java-bindings layer

The Java-bindings layer handles the connection from the Java-code to the PEIS-kernel C-code via the JNI-code that this layer contains. This layer is not totally separated from the next Java layer that bring object orientation to the code, but it is quite easy to separate from the Java object layer. The design of this layer is straightforward. The Java-code include native declarations which can look like this native void foo(); where the keyword native is used to show that it is a native function (C-function). The foo-function can be used like any other Java-function. When called a JNI-code function is accessed. The interface-code functions can look like this JNICALL \texttt{Java\_PEIS\_JavaJNI\_Tuple\_foo(JNIEnv *env, jobject obj) \{ peisk\_foo(); \}} . JNI-EXPORT and JNICALL are macros that are needed to define the linkage and calling convention for the native functions and JNI functions. The much longer function name in the JNI-code for the foo function defined in Java
3.2. JAVA-BINDINGS LAYER

Java_PEISJavaJNI_Tuple_foo defines the structure of from where the JNI function is called. PEISJavaJNI is the package that include the file and class that contain the native function declaration and calls the JNI function. In the JNI-code function the calls to the native function peisk_foo() is made. An important point of having the Java-bindings layer is that the code is easier to use with different projects. The native function declarations must otherwise be put in any Android application and used directly from that application. The negative side effect when placing the native function declarations in the Java application is that the function name in the JNI-code file JNIEXPORT void JNICALL Java_PEISJavaJNI_Tuple_foo needs to be changed to be used in different projects (if they have different names for the class that call the JNI-code function). The code would be much harder to use in that case. This layer is equivalent to PEIS-JavaJNA. If this layer would be used it would be written like this:

Subscription to a tuple

pk_subscribe(name, owner);

Publish a tuple

pk_setTuple(name, data_length, data);

Get a tuple

tuple = pk_getTuple(name, owner);

Basically just function calls like in C-code. It should be noted that this layer is not accessible for developers as a security measure, it is only shown here as a comparison to the other layers. The main contributions from the Java-bindings layer is that it gives a simple way to access the JNI-code functions and that it makes PEIS-JavaJNI easier to use. One reason that this layer is not accessible to developers of PEIS-components is that PEIS-JavaJNI is supposed to work with multi-threaded programs in which calls can come from multiple directions to many different functions at the same time. This is many times the case with modern programs as the program execution is not tied to a single so called thread of execution. Multiple threads can be used in a program to deal with different parts in the program. An example of this would be a program that got a thread to survey a tuplespace, another that handles the tuples that are found and a third that publish tuples based on the tuples that are found in the tuplespace. Multi-threaded programs can get problems when functions for writing and reading are used at the same time. A read function might have read half the fields of a tuple when a write function changes the rest so the mix of fields do actually not exist. In the next layer Java object layer this is handled with so called synchronization blocks that only allow one thread of a program to execute in a certain code section.
3.3 Java object layer

The Java object layer adds a much better interface for developers than just raw access to the native functions that the Java-bindings layer provide. This layer adds object orientation and in this also hides the underlying C-code complexity. Another benefit of this layer is that it gives more secure access to the PEIS-kernel code as the C-code is not accessed as direct and dangerous mistakes can be avoided as it is clear what can be done with certain objects. The java object layer is designed to be object oriented. This means that each large collection of functions and functionality is grouped under what is called a class. An example of this is the PeisTuple object which represents a tuple. It is only possible to create a PeisTuple object with predefined functions and to use the tuple with functions that are defined for the PeisTuple object. The template that describes what an object holds for data and what functions it has is called a class. The class is a template and when an instance of a class is created (like a variable) the result is an object of the class type. A class have many useful properties as it can inherit functionality from another class. This is for example used for this work for the classes PeisTuple, AbstractTuple and MetaTuple that inherits from a class called Tuple. Without the inheritance from the Tuple class much of the same functionality would have had to been written in each of these three (3) classes. The Java object layer consist of ten (10) classes that can be seen in figure 3.2.

PeisJavaMT: Contains functionality that is general to the PEIS-kernel, as initialization, stepping, making it wait, halting and getting error information. This class also contains an inner class that lets the PEIS-kernel be stepped in its own thread, this inner class is called PeiskThread.

PeiskThread: An internal class of the class PeisJavaMT. This class automatically steps the PEIS-kernel, checks for PEIS-kernel errors and runs/updates the callback handler in the class PeisJavaMT. This code in this class will run in a thread that is separate from the main application. This is done so that the PEIS-kernel will run separate from the application that uses it. This is very helpful because the stepping function that makes the PEIS-kernel run does in this way not need to be called explicitly from the main application. The gained benefit from the separate thread is that the main application do not need to wait while the PEIS-kernel is stepped. This class is not used by PEIS-component developers. The separate thread with an instance of the class PeiskThread is created and started when the function to initialize the PEIS-kernel is called.

Tuple: The base template for all tuple types. It contains functionality to create and use a tuple on the basic level. The three (3) other tuple types inherits the functionality from this class and adds functionality specific to that tuple type.
AbstractTuple: Describes the tuple type abstract tuple that is used when a simple tuple prototype with name and owner is needed as for subscriptions. This class contains functionality to create and initialize an abstract tuple and to test if it is an abstract tuple.

PeisTuple: Describes a normal tuple with data. This class contains functionality to create and initialize a normal tuple.

MetaTuple: Describes the meta tuple type that is used for indirect tuple operations, as a meta tuple works as a pointer to another tuple. This class contain functionality to create and initialize a meta tuple. It also contains functions to write data and get data through meta tuples.

Subscription: Describes a subscription to tuples. It contains functionality to create, register and unregister a subscription to a tuple. It also contains functionality to get the tuple(s) that match a subscription.

SubscriptionCB: Describes a subscription to a tuple that also incorporate a callback functionality. This class inherits the functionality from the Subscription class. As a callback for a tuple is registered with the PEIS-kernel, the PEIS-kernel will keep track of when there is a change in that specific tuple and call back to the Java code. The call is done to a specified callback function in the SubscriptionCB class for each SubscriptionCB object.

PeisCallbackHandler: Is used to handle callbacks, an instance/object of it is called a callback handler. A callback works by activating a specific function (defined by the programmer) when a tuple in the tuplespace is changed. In PEIS-JavaJNI callbacks are represented by SubscriptionCB objects. All SubscriptionCB objects are stored in an instance/object of the PeisCallbackHandler. The callback handler updates which callbacks to add and remove, a program thread can only register that it wants to add a new callback or ask the callback handler to remove the callback. The actual registration and removal is done by the callback handler. This is done so that callbacks work in a multi-threaded environment.

WrongPeisKernelVersionException: Describes an exception that is triggered when the PEIS-kernel is about the be started. If the PEIS-kernel version do not equal the PEIS-JavaJNI version.

To accomplish the basic tasks for a PEIS-component is done in this way:

Subscribe to a tuple

```
Subscription subs = new Subscription(new AbstractTuple(name));
```

Publish a tuple

```
PeisTuple tuple = new PeisTuple(name, data, mime_type);
tuple.insert();
```
Figure 3.2: These are the classes in software layer 3 that is the Java object layer in PEIS-JavaJNI.

Get a tuple from the tuplespace

```java
Tuple tupleResult = subs.getTuple();
String result = tupleResult.getData();
```

These examples show a clear difference to the example with the underlying layer, the Java-bindings layer. The examples show how the functionality is divided to different objects as the subscription object. The benefits to use this layer over the underlying Java-bindings layer is that it is more clear what can be done in a certain context as the functionality is divided to different objects. Another important aspect of this layer is that it adds a multi-thread safe environment. This means that it is safe to use this layer with multiple so called threads. A thread is a piece of code that execute on its own as another program. In the current implementation of PEIS-JavaJNI, two (2) threads are used. One thread for the PEIS-kernel and one for the application (unless the application need more than one thread). It is made multi-thread safe by the use of so called synchronization blocks that look like this:

```java
synchronized(object) {
}
```

The synchronization blocks do so that only one thread at the time can run the code inside any of the blocks. This is done by having an object that works like a ticket for a ride that only take one person (thread) at the time. Once the thread leaves the block it returns the control of the object so another thread can enter a block of code tied to the object.

### 3.4 Android abstraction layer

The last layer before the Android application is either the Java object layer or the Android abstraction layer. The Android abstraction layer holds Android
specific code that is useful for development of PEIS-components for an Android system. The code for this layer is in \textit{PEIS-Android}. PEIS-Android currently contains one class called \textit{PeisAndroid} that contains one function, called \textit{sendWirelessEnabled}, that publish a tuple with a string that says \textit{enabled} or \textit{disabled} based on the WiFi-state. To know if the WiFi is turned on can be very useful as the latest generation of the PEIS-kernel (generation 6) can communicate via Bluetooth. So it makes sense to have some way to ask the device if it can communicate with WiFi. The benefits to use the functions in this layer is that there is no need to re-implement the needed functionality to use Android specific functionality.

### 3.5 UML sequence diagrams

The basic tasks a PEIS-component must achieve are subscribing to a tuple, publishing and retrieving tuple. To get a view of how the code is designed and how it works a few UML sequence diagrams are included. Figure 3.3 shows a sequence diagram of what happens in the program for a subscription to the tuple. First the constructor in the Subscription class is called then the PEIS-kernel code for a subscription is called through the JNI-code layer. Then the execution returns to the application with a new subscription object. Even though the subscription is registered with the PEIS-kernel when the execution returns, it does not mean that the subscription has propagated in the PEIS-network yet. The reason for this is that the PEIS-kernel runs asynchronous from the Android application and it can take time for the subscription to be processed and handled by the different layers of communication that is used in a PEIS-network.

![Sequence Diagram](image)

**Figure 3.3:** Sequence diagram that shows what happens when a subscription is issued from an application. The boxes represents sections of code and the arrows represents calls to functions in those code blocks.
Figure 3.4 shows the sequence diagram for publishing a tuple. The diagram shows how a tuple with a string as data is published. First the setStringTuple() function in the class Tuple is called which in turn call the PEIS-kernel function to publish a tuple with a string as data. Then the execution returns to the application. As with the subscription call it is not certain that the tuple is propagated in the PEIS-network when the execution returns to the Android application.

![Sequence Diagram for Publishing a Tuple](image)

**Figure 3.4:** Sequence diagram that shows what happens when a tuple is published from an application. The boxes represents sections of code and the arrows represents calls to functions in those code blocks.

Figure 3.5 shows the sequence diagram for retrieving all the tuples that match the subscription. The sequence diagram also shows how the owner, key and data is retrieved from a tuple that is returned. First the function getTuples() in the class Tuple is called and then the PEIS-kernel function to get all matching tuples are called through the JNI-code layer. The PEIS-kernel function returns all tuples in a special format called a PeisTupleResultSet that is transformed into an array of tuple objects by the JNI and Tuple class code. The array of tuple objects is returned to the application. The next part in the sequence diagram show what happens when owner, key and data is retrieved from one of the tuples in the array of tuples. The procedure is the same in all the three (3) cases, first the functionality in the Tuple class is called and then the JNI-code is called. No call to PEIS-kernel code is needed as the JNI-code can handle the extraction of tuple information and data itself. The information or data in the tuple is returned from the JNI-code to the Tuple class function which in turn returns it to the application.

![Sequence Diagram for Retrieving Tuples](image)
3.6 Example application: getTuple

One of the most frustrating parts of leaping into a new field is to not know where to start. That is why the example application `getTuple.java` has been developed to show in a clear way how to do basic things with PEIS-JavaJNI and PEIS-Android. The application show how to subscribe, publish and get tuples from a PEIS-network. Figure 3.6 show the interface for the getTuple application. The application can publish two tuples with the name `atuple1` and `atuple2` as well as 100 tuples that are named `btupleX` where X is the number of the tuple. The application can search and retrieve two tuples `atuple1` and `atuple2` as well as search for and retrieve all tuples with any name and with any
Figure 3.6: Interface of the example application getTuple, that is supposed to be a start for developers of PEIS components for Android.

owner. In the figure of the interface, tuple1 and tuple2 have been found and the tuples owner, name and data is displayed. At the bottom of the interface the result of searching and retrieving all available tuples in the tuplespace is shown.
Chapter 4
Evaluation

This chapter covers the tests and evaluations for the resulting software, the developed API. This is done to evaluate if the resulting code has reached a good level of maturity and how easy or hard it is to use and get started using the API. The evaluation will also show if the different goals and objectives for this work has been reached.

4.1 Static tests

4.1.1 Metrics

The program used to extract metrics information is the Eclipse metrics plug-in from State Of Flow [26]. Which give a large range of values for the Java-code. The reason that this evaluation is done is that it will show many valuable statistics about the code. One example is the metric cyclomatic complexity that can show where the code complexity is high and where many tests are needed. If this value is very high for a section it can be a good idea to look into if the section could be rewritten and divided into smaller code sections in separate functions. A high complexity can also hide errors that are hard to find and it gets more difficult to maintain a more complex code.

Results

The overall results can be seen in table 4.1. The reason to why there are three packages is that the example application is in a package for itself. Table 4.2 show a number of metric values for PEIS-JavaJNI and PEIS-Android. The following list is an explanation to the mentioned metrics values in table 4.2:

CC:  *Cyclomatic Complexity*. This measures the number of branches in the code. The more ways that execution can go through the code the more tests are needed to cover all possible cases and also this is a measure for complexity of the code. A low value for this measure means that there
Table 4.1: This table show the overall results for PEIS-JavaJNI and PEIS-Android, and the test application getTuple.

<table>
<thead>
<tr>
<th>Packages</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>12</td>
</tr>
<tr>
<td>Total lines of code</td>
<td>1995</td>
</tr>
</tbody>
</table>

are few execution paths through a code section which is good out of a debugging and maintenance viewpoint. The measure is done for methods.

**LOCm:** *Lines of Code in Method.* This shows the number of lines in a method. This is not a very good measure as comments and empty lines are included.

**NLS:** *Number of Locals in Scope.* Indicates the number of variables that are in scope (aka, locals) at any time in a method. The scope of a variable is the block of code within which the variable is accessible (e.g., a for loop, a method in Java, a function in C, etc.). A large number can point to that the code should be split up into more methods to decrease the complexity so not that many variables are in scope at once.

**NOL:** *Number of Levels.* This show the level of nesting in a method. A large number can indicate a high complexity and that it could be a good idea to divide the method into more methods.

**NOP:** *Number of Parameters.* Show the number of parameters to a method. A large number can indicate that classes are missing from the overall design. A solution can be to make classes that contain some of the parameters.

**NOS:** *Number of Statements.* Show the number of statements in a method. A large number can indicate that some of the functionality could be moved to separate methods.

**NOF:** *Number of Fields.* Show the number of fields in a class. A large number can point to that the class can be divided into more classes.

**WMC:** *Weighted Methods Per Class.* This is the sum of all cyclic complexity numbers for the methods in a class.

Table 4.2 show that there is a clear difference between PEIS-JavaJNA and PEIS-JavaJNI. The metrics data show that PEIS-JavaJNI has a lower level of complexity and a more streamlined architecture than PEIS-JavaJNA. The cyclic complexity (CC) is lower for the PEIS-JavaJNI which would suggest a more maintainable code as the debugging is not as hard and the code is hopefully easier to understand. Two numbers are however larger for the PEIS-JavaJNI when compared to the PEIS-JavaJNA. The lines of code in method (LOCm),
4.1. STATIC TESTS

Table 4.2: This table show the overall metrics results for PEIS-JavaJNA and the developed PEIS-JavaJNI and PEIS-Android.

<table>
<thead>
<tr>
<th>Package</th>
<th>CC</th>
<th>LOCm</th>
<th>NLS</th>
<th>NOL</th>
<th>NOP</th>
<th>NOS</th>
<th>NOF</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEIS-JavaJNA</td>
<td>15</td>
<td>67</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>48</td>
<td>16</td>
<td>153</td>
</tr>
<tr>
<td>PEIS-JavaJNI</td>
<td>11</td>
<td>86</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>55</td>
<td>5</td>
<td>123</td>
</tr>
<tr>
<td>PEIS-Android</td>
<td>2</td>
<td>39</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3: Metrics values for the individual classes in PEIS-JavaJNI.

<table>
<thead>
<tr>
<th>Class</th>
<th>CC</th>
<th>LOCm</th>
<th>NLS</th>
<th>NOL</th>
<th>NOP</th>
<th>NOS</th>
<th>NOF</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PeisJavaMT</td>
<td>11</td>
<td>86</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>55</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>Tuple</td>
<td>6</td>
<td>44</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>22</td>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>PeisCallbackH...</td>
<td>3</td>
<td>27</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>PeiskThread</td>
<td>3</td>
<td>18</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MetaTuple</td>
<td>4</td>
<td>33</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>15</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>AbstractTuple</td>
<td>1</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>PeisTuple</td>
<td>1</td>
<td>19</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Subscription</td>
<td>4</td>
<td>27</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>SubscriptionCB</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>WrongPeisK...</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

this is from the start a bad measure of the code as the line count also includes comments and empty lines. This is the case as PEIS-JavaJNI has quite many comments to clarify the usage and also a more spread out layout of the code to provide a less compact code that is easier to read. The second number that is higher for PEIS-JavaJNI is the number of statements in a method (NOS), the difference is not large and it is for one method in the class PeisJavaMT (see 4.3) that handles the initialization of the PEIS-kernel. One conclusion that can be drawn from these numbers is that quite much complexity is hidden in PEIS-JavaJNI that the developer do not need to worry about. Table 4.2 also show the metrics for PEIS-Android, as it does not contain much the numbers are quite low. Table 4.3 show the metrics for the individual classes in PEIS-JavaJNI, and Table 4.4 show the metrics for the class in the PEIS-Android and the class that include the example application getTuple. A class by class comparison between PEIS-JavaJNA and PEIS-JavaJNI would not give much as they use different classes, that is why only the overall numbers for PEIS-JavaJNA and PEIS-JavaJNI can be compared. Table 4.3 show that the highest complexity is concentrated to the classes PeisJavaMT and Tuple. This is not a surprise as PeisJavaMT contain the initialization for the PEIS-kernel as well as the inner class PeiskThread. PeiskThread handle the separate thread for the PEIS-kernel that steps the PEIS-kernel and check for errors. The class Tuple is the base class
Table 4.4: This table show the metrics values for the class in PEIS-Android and for the class in the example application getTuple.

<table>
<thead>
<tr>
<th>Class</th>
<th>CC</th>
<th>LOCm</th>
<th>NLS</th>
<th>NOL</th>
<th>NOP</th>
<th>NOS</th>
<th>NOF</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTuple</td>
<td>4</td>
<td>42</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>18</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>PeisAndroid</td>
<td>2</td>
<td>39</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

for all tuple types and hold all necessary functionality for the basic tuple so it makes sense that it would have a higher complexity and size than the classes further down in the table. The reason that the example application getTuple get such a high place for cyclic complexity in table 4.4 is that it contain functions that tests subscriptions and tuples.

4.1.2 PEIS-Android

PEIS-Android and the class PeisAndroid that it contains, correspond to objective 4. The functionality this layer currently holds is a function to check if the WiFi is enabled and publish a tuple with the result. The goal for for objective 4 was to provide the structure and possibility to gather and use Android specific functionality in one place. Even though PEIS-Android does not contain more than one example, it shows how to include more functionality. And thus fulfills the goal of the objective 4, there is much that can be done with this layer but that is left for future work [Section 5.2].

4.2 Performance tests

One very important factor in evaluating the resulting code, is how well it performs during the common tasks. That is why the following tests have been performed on an Android application performing subscription, publication and retrieval of tuples from the tuple-space.

General test conditions: The PEIS-components have access to a wireless network that is only used by one other PEIS-component (tupleView, a program to survey a tuplespace) other than the PEIS-components that are tested. There are two PEIS-components used:

\( C_{HTC} \)

- **Hardware**: Executes on a mobile phone (HTC Desire HD) with Android version 2.3.3 and a 1 GHz processor.
- **Software**: PEIS-JavaJNI that was developed in this work is used for the PEIS-component.
4.2. PERFORMANCE TESTS

CPC

- **Hardware:** Executes on a computer with a dual-core processor running at 2.30 GHz.

- **Software:** PEIS-JavaJNA is used for this PEIS-component.

The measurements are the time it takes to perform certain task and the given values in the tables are an average of 5 tests that are rounded to whole milliseconds.

**Test 1:** Subscription to tuples.
**Test description:** The PEIS-components issues a subscription to tuples from any owner and with the name structure of atupleX, where X is the number of the tuple that is handled (from 1 up to 10000). The time that is measured is only the time it takes to make the call to the PEIS-kernel which does not include the time for the subscription to propagate in the PEIS-network. This is so that the comparison only measures the difference between PEIS-JavaJNI and PEIS-JavaJNA. Differences in how the devices are connected to the PEIS-network, the performance of their network interfaces, as well as dynamic network conditions, are therefore removed from the test.

**Results:** The Android application that uses PEIS-JavaJNI performs faster for groups of subscriptions up to 2000 tuples but is almost three (3) times slower for groups of subscriptions with around 10000 tuples. The time results for subscription to different number of tuples can be seen in table 4.5. The results can be seen in a graphical form in plot 4.1. This plot show how the execution time for the two components increase more and more, and the increase is not linear to the number of tuples that the components subscribe to. This plot clearly show how CHTC is faster for groups of subscription up to 2000 tuples but after that become slower than CPC. The plots also show that the two components behave in a similar fashion. Although the execution time increase much faster for CHTC than for CPC. The large difference could be the result of that objects are created. To test this theory a complementary test for subscriptions without creating any specific subscription objects for a subscription was tested to have no significant influence. If in the Android application the creation of subscription objects was skipped, the subscription to 10 000 tuples took in average 45565 ms, which is a slight decrease in execution time but not close to the execution time of CPC for the same task. That means that the object oriented design is not the cause of the much slower execution times.

**Test 2:** Publishing tuples.
**Test description:** The PEIS-components publish a number of tuples containing a text string. The tuple name structure is atupleX, where the X is the tuple number (1 to 10000). The measured execution time is the time it takes to call
Table 4.5: Time measurements in milliseconds for execution of subscribing to groups of tuples. C$_{HTC}$ runs on a mobile phone (HTC Desire HD) and use the developed PEIS-JavaJNI, C$_{PC}$ runs on a computer with a dual core processor running at 2.30 GHz and use PEIS-JavaJNA.

<table>
<thead>
<tr>
<th>Subscriptions</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>7000</th>
<th>9000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>C$_{HTC}$</td>
<td>&lt;1</td>
<td>1</td>
<td>11</td>
<td>82</td>
<td>370</td>
<td>1782</td>
<td>12177</td>
<td>23422</td>
<td>39537</td>
<td>48650</td>
</tr>
<tr>
<td>C$_{PC}$</td>
<td>7</td>
<td>10</td>
<td>62</td>
<td>331</td>
<td>663</td>
<td>1687</td>
<td>5389</td>
<td>9012</td>
<td>13477</td>
<td>16656</td>
</tr>
</tbody>
</table>

Figure 4.1: This plot show the execution time for a subscription to a number of tuples with a logarithmic scale for the time. C$_{HTC}$ runs on a mobile phone (HTC Desire HD) and use the developed PEIS-JavaJNI, C$_{PC}$ runs on a computer with a dual core processor running at 2.30 GHz and use PEIS-JavaJNA.

the PEIS-kernel, it does not include the time for the tuple to be propagated by the PEIS-kernel to the network. This is so that the comparison only measures the difference between PEIS-JavaJNI and PEIS-JavaJNA. Differences in how the devices are connected to the PEIS-network, the performance of their network interfaces, as well as dynamic network conditions, are therefore removed from the test.

Results: The time measurements that can be seen in table 4.6 show that C$_{HTC}$ can not quite keep up with C$_{PC}$, it is a bit slower for every group of tuples that is published. The difference in time to publish 10000 tuples with C$_{HTC}$ and publishing with C$_{PC}$ is almost 60 seconds, a big difference but it is only about 2.5 times slower. Plot 4.2 show the measurements for execution time for the number of tuples that are published. In this plot the same behavior as in test 1 can be seen. The execution time to accomplish the task for C$_{HTC}$ and C$_{PC}$ increase more and more. In this plot it can be seen that the two components behave in a similar fashion as to execution time over number of tuples published.
4.2. PERFORMANCE TESTS

Table 4.6: Time measurements in milliseconds for execution of publishing tuples. C\textsubscript{HTC} runs on a mobile phone (HTC Desire HD) and use the developed PEIS-JavaJNI, C\textsubscript{PC} runs on a computer with a dual core processor running at 2.30 GHz and use PEIS-JavaJNA.

<table>
<thead>
<tr>
<th>Publication</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>7000</th>
<th>9000</th>
<th>10 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component1</td>
<td>&lt;1</td>
<td>3</td>
<td>33</td>
<td>298</td>
<td>1031</td>
<td>4163</td>
<td>25027</td>
<td>48570</td>
<td>79518</td>
<td>97735</td>
</tr>
<tr>
<td>Component2</td>
<td>&lt;1</td>
<td>1</td>
<td>14</td>
<td>123</td>
<td>411</td>
<td>1543</td>
<td>9641</td>
<td>18871</td>
<td>31165</td>
<td>38322</td>
</tr>
</tbody>
</table>

Figure 4.2: This plot show the execution time for publishing a number of tuples with a logarithmic scale for the time. C\textsubscript{HTC} runs on a mobile phone (HTC Desire HD) and use the developed PEIS-JavaJNI, C\textsubscript{PC} runs on a computer with a dual core processor running at 2.30 GHz and use PEIS-JavaJNA.

The relationship between the execution times is pretty much constant over all the groups of tuples that are published. C\textsubscript{HTC} is about 2.5 times slower than C\textsubscript{PC} no matter the number of tuples that are published.

**Test 3:** Tuple retrieval.

**Test description:** The PEIS-components subscribe to tuples from any owner and with any name. The components also publish the same tuples that are retrieved and owner, key and data is extracted. Only the time to retrieve the published tuples is measured. The components themselves publish the tuples that they retrieve so that the tuples are in the local tuplespace for the PEIS-component. This is so that the comparison only measures the difference between PEIS-JavaJNI and PEIS-JavaJNA. Differences in how the devices are connected to the PEIS-network, the performance of their network interfaces, as well as dynamic network conditions, are therefore removed from the test.
Table 4.7: Time measurements in milliseconds for execution of retrieving tuples. \( C_{HTC} \) runs on a mobile phone (HTC Desire HD) and use the developed PEIS-JavaJNI, \( C_{PC} \) runs on a computer with a dual core processor running at 2.30 GHz and use PEIS-JavaJNA.

<table>
<thead>
<tr>
<th>Retrieval</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
<th>7000</th>
<th>9000</th>
<th>10 000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{HTC} )</td>
<td>4</td>
<td>5</td>
<td>31</td>
<td>145</td>
<td>221</td>
<td>423</td>
<td>1051</td>
<td>1526</td>
<td>1961</td>
<td>2181</td>
</tr>
<tr>
<td>( C_{PC} )</td>
<td>2</td>
<td>9</td>
<td>60</td>
<td>260</td>
<td>484</td>
<td>748</td>
<td>961</td>
<td>1195</td>
<td>1308</td>
<td>1461</td>
</tr>
</tbody>
</table>

Figure 4.3: This plot show the execution time to retrieve a number of tuples with a logarithmic scale for the time. \( C_{HTC} \) runs on a mobile phone (HTC Desire HD) and use the developed PEIS-JavaJNI, \( C_{PC} \) runs on a computer with a dual core processor running at 2.30 GHz and use PEIS-JavaJNA.

**Results:** The time measurements that can be seen in table 4.7 show that the difference in execution time is not as high as in the two other tests. The two components are quite close in execution time for all groups of tuples that are retrieved. The difference between the two components for retrieving a group of 10000 tuples is 720 milliseconds, a small difference for such a large task if compared to the other tests. Plot 4.3 shows graphically how \( C_{HTC} \) and \( C_{PC} \) are close in execution time and that \( C_{HTC} \) is faster than \( C_{PC} \) for retrieval of groups of tuples that have about 5000 tuples or less. The plot also shows how small the difference is in execution time between the two components when compared to the two other tests. The two components also in this test, as in test 1 and test 2, have a similar behavior when it comes to the execution time for handling tuples.

**Evaluation of tests:** \( C_{HTC} \) is faster or not significantly slower when handling
groups of tuples that are smaller than 2000 tuples when compared to \( C_{PC} \). \( C_{HTC} \) executed on a mobile phone and used the developed PEIS-JavaJNI and \( C_{PC} \) executed on a computer with a dual-core processor running at 2.30 GHz and used PEIS-JavaJNA. The important result from these tests is that a PEIS-component running on a mobile device and uses PEIS-JavaJNI performs very well when compared to a PEIS-component that runs on a computer, even if the computer is much faster, and uses PEIS-JavaJNA. As the tests show there do not seem to be a linear relationship between the number of tuples handled and the execution time for the components. What this is a result of is hard to say. It could be the result of how the PEIS-kernel handles tuples, how the virtual machines (Java virtual machine for \( C_{PC} \) and Dalvik virtual machine for \( C_{HTC} \)) handles memory and execution, or how data is handled when using JNA (\( C_{PC} \)) or JNI (\( C_{HTC} \)). What is important however is that \( C_{HTC} \) was able to perform at a good level even though it executed on a mobile device. This show that PEIS-JavaJNI can be used practically for PEIS-components.

Another comparison that can be made is between two PEIS-component that are tested on the same computer, where one PEIS-JavaJNA and the other PEIS-JavaJNI. In this test the two components would execute on the same hardware. This test would compare PEIS-JavaJNA and PEIS-JavaJNI on more equal terms. Because of time constraints this test is left for future work with PEIS-JavaJNI.

### 4.3 Development: PEIS-JavaJNI vs. PEIS-JavaJNA

PEIS-JavaJNA that was used to create PEIS-components in Java used JNA to access the PEIS-kernel C-code. It was basically a wrapper for the PEIS-kernel functionality and only offered a few additions over the PEIS-kernel functionality. It have some objects and also have support for multi-threading. PEIS-JavaJNI use JNI and can be used for both computers and Android systems. The performance between these two are of interest to see if it is a good idea to use PEIS-JavaJNI for better efficiency.

#### 4.3.1 Size comparison

A simple application written with the use of PEIS-JavaJNA that publishes a tuple takes up almost 2 KB, PEIS-JavaJNA about 30 KB and the PEIS-kernel library about 400 KB which make a total of 432 KB. The written getTuple Android application that use PEIS-JavaJNI and PEIS-Android take up in total 460 KB installed. It subscribes, publish, retrieves tuples and presents the tuples information on the screen of the mobile device. So if the memory space available is not very limited this is not a factor. Also the test in the previous section (Live tests) show that the execution speed of a Java PEIS-component that use PEIS-JavaJNA and and executes on a computer with a 2.30 GHz dual-core processor...
can be matched on many tasks by a PEIS-component that use PEIS-JavaJNI and executes on a mobile phone as the HTC Desire HD.

### 4.3.2 Usability comparison

To investigate the difference between PEIS-JavaJNI and PEIS-JavaJNA a typical PEIS-component will be created using each of the Java packages. To highlight the differences only the main code-differences are shown. The PEIS-components publish two (2) tuples, subscribes to tuples with any owner and any name. And finally retrieves all tuples with any name and any owner from the tuplespace and extract the tuples owner, key/name and the data as a string.

**PEIS-JavaJNA**

// Publish two tuples.
PeisJavaMT.peisjava_setStringTuple("atuple1", "atuple1_msg");
PeisJavaMT.peisjava_setStringTuple("atuple2", "atuple2_msg");

// Subscribe to all tuples.
PeisJavaMT.peisjava_subscribe(-1, "*");

// Get all tuples from the tuplespace.
PeisTupleResultSet res = PeisJavaMT.peisjava_getTuples(-1, "*");
PeisTuple tuple = res.next();

// Go through found tuples and extract tuple owner, key and data.
while(tuple != null) {
    tuple.owner;
    tuple.getKey();
    tuple.getStringData();

    tuple = res.next();
}

**PEIS-JavaJNI**

// Publish two tuples.
PeisTuple.setStringTuple("atuple1", "atuple1_msg");
PeisTuple.setStringTuple("atuple2", "atuple2_msg");

// Subscribe to all tuples.
Subscription sub_atuple1 = new Subscription(-1, "*");

// Get all tuples from the tuplespace.
Tuple[] results = sub_atuple1.getTuples();
4.4. DEVELOPMENT TESTS

<table>
<thead>
<tr>
<th>Class</th>
<th>CC</th>
<th>LOCm</th>
<th>NLS</th>
<th>NOL</th>
<th>NOP</th>
<th>NOS</th>
<th>NOF</th>
<th>WMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>testApp1</td>
<td>2</td>
<td>18</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>testApp2</td>
<td>5</td>
<td>29</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>25</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

What is important to notice for PEIS-JavaJNI in this comparison is the simpler and cleaner way of doing the common PEIS-component tasks of subscribing, publishing and retrieving tuples. The functions to use are not gathered and accessed through one class (PeisJavaMT) as with PEIS-JavaJNA but is divided to specific classes as the Tuple class and the Subscription class. This division of functionality makes it conceptually easier for the user and helps when trying to find a certain function and to use the API (PEIS-JavaJNI). As an example of the more simpler interface is that the PeisTupleResultSet in PEIS-JavaJNA is not used by the user. Instead an array of tuples is returned to the user. This comparison shows that objective 3 to make it easier to develop PEIS-components with a Java object layer is achieved. As objective 3 is achieved also objective 1 to be able to run a PEIS-component on an Android device and objective 2 to abstract away the calls to the C-functions is achieved.

Metrics for PEIS-components: The metrics for these two test applications can be seen in table 4.8. The application testApp1 use PEIS-JavaJNI and testApp2 use PEIS-JavaJNA.

4.4 Development tests

The API is written to be easily understood and easy to use. With the use of the getting-started guide (appendix A) the goal is that any Java and Android applications developer should be able to include the PEIS-ecology functionality as easily as possible. A real test of how and if a Java and/or Android applications developer would be able to use the API would show if the secondary goal has been reached. The secondary goal was to provide a clear, concise and easy way for Java and Android application developers that was not familiar with development of PEIS-components to integrate PEIS-ecology functionality in their applications. To test if this goal has been achieved a task that included
the use of PEIS-functionality was given to an Android applications developer. The developer had not previously worked with PEIS-component development so this would also test the use of the getting-started guide (appendix A) and the documentation for PEIS-JavaJNI.

**Test environment:** The test of the API is performed on a computer with PEIS-JavaJNI and PEIS-Android. Eclipse is used for the test. A new project for Android applications is provided in Eclipse, without any PEIS-functionality implemented. The provided project do however include the JNI and PEIS-kernel code in its directory structure.

**Test description:** The tester is given the task to create an application for the Android system that includes the functionality to subscribe to, publish and retrieve tuples. The help for this task is the documentation for PEIS-JavaJNI and PEIS-Android and the getting-started guide. To get a better perspective on what weak spots there are in using the API the developer is also asked to estimate how much time it took to (1) understand the basic idea of PEIS, (2) include subscription, (3) include publishing and (4) include retrieval and extraction of tuples and tupledata. The tester is asked to evaluate the use of the getting-started guide and the documentation and put a number between 1 and 5 on how easy or hard it was (1 is easy, 5 is very hard) to accomplish the different steps by using the documentation and the getting-started guide.

**Results:**

- **Understand PEIS-basics:** Time: 20 minutes, difficulty: 3.
- **Include subscription:** Time: 2 minutes, difficulty: 1.
- **Include publishing:** Time: 2 minutes, difficulty: 2.
- **Include retrieval:** Time: 4 minutes, difficulty: 3.
- **General notes from the tester:** 15-20 min is enough to understand the main structure in the code and figure out the order functions should be placed in the code. The whole documentation is well-structured and provides the most essential information required for basic understanding of the system. If a Java programmer starts operating with PEIS-kernel environments without any preliminary knowledge, the user-guide can be a perfect introduction for a good start.

This test shows that it does not take much time to incorporate basic PEIS-functionality in an Android application. The difficulty is not that high either and is at a medium level of 3 as the highest and most difficult level. The first difficult part (medium difficulty) for the tester was to understand the basics for a PEIS-system. This part explains how the basic PEIS-functionality works and also how it can be implemented. It is not surprising that this part got a high
mark as the tester was unfamiliar with the PEIS-idea. The second part that got a high rating for difficulty was to include tuple retrieval from a tuplespace. This part also incorporate the extraction of name, owner and data from a retrieved tuple. So the tester needed to use more functions from the Java package to complete this part. Overall it takes less than half an hour to understand the basic idea for a PEIS-system and incorporate basic functionality into an Android application.

4.5 Current limitations

Almost all the functionality that the PEIS-kernel can provide is possible to access and use with PEIS-JavaJNI. What is not working at the moment is to register so-called callbacks for tuples. Callbacks provide the functionality to automatically get an activation of a function on the user side when a tuple in the tuplespace is changed. At the current state of the code a callback can be registered but the activation of the function on the user side when a tuple is changed gets stuck in the JNI-code. This problem is believed to be caused by a problem to locate the correct object and function in the Java code that should be called from the JNI-code. The structure for creating and registering a callback is implemented but the last crucial steps to close the circle is not working. This problem is most likely possible to solve, but some time is needed.

4.6 Summary

The developed PEIS-JavaJNI works very well for PEIS-components when compared to the existing PEIS-JavaJNA. A PEIS-component that use the PEIS-JavaJNI and executes on a mobile device as the HTC Desire HD, can on many tasks match a PEIS-component that use PEIS-JavaJNA and executes on a 2.30 GHz computer. This is quite impressive as the limited resources of the mobile device is much smaller than that of the computer. The new API (PEIS-JavaJNI) has been tested to be simple to use. With the getting-started guide (appendix A and the documentation for the API, a Java and Android applications developer that do not have previous knowledge of the PEIS idea can include PEIS-functionality without much struggle. The overall result of the evaluation is that all objectives for this work has been achieved.
Chapter 5
Conclusion

Android is a powerful base for development as it provides many different tools and the devices that run it in many cases provide a large set of hardware that can be used for a PEIS-component. One part of this work was to gather all the knowledge gained for this work that was not directly tied to the developed code but more to the operational aspects of Android and JNI development. This is so that the next developer of PEIS-components for Android can get started with more than just the resulting code but also the basic knowledge of how to get started and possible pitfalls to avoid. This resulted in Appendix A that explains what is needed and how to set up to get things to work. It also give some pointers that can be useful for development.

My impression of the Android system and development for it is a positive one. There are many things to learn initially when starting development but Google provides a very good documentation and there exist quite many good on-line guides and forums where many problems have already been solved. For reading material that do not require internet access a good and easily understood book is Hello Android [7] or Beginning Android 2 [27], for more advanced topics Pro Android 2 [24] can be a good source for information. To get a good start in JNI-development Sheng Liangs book The Java Native Interface, Programmer’s Guide and Specification [21] is a good choice.

5.1 What does this work provide

This work has made it possible to develop PEIS-components for Android devices with a Java package that can also be used to develop PEIS-components for computers. There is a big added value of having one package that can be used with both Android devices and computers as improvements and debugging only needs to be done in one package.

The developed PEIS-JavaJNI is faster for many tasks than the existing solution
CHAPTER 5. CONCLUSION

PEIS-JavaJNA. A PEIS-component that executes on a mobile device like the mobile phone HTC Desire HD with a 1 GHz processor and uses PEIS-JavaJNI can even be faster than a PEIS-component that executes on a computer with a dual-core 2.30 GHz processor and uses PEIS-JavaJNI.

With this work it is now possible to have humans as a much more integral part of a PEIS-ecology if a mobile device with Android is used. This gives many more possibilities for PEIS-ecologies as the ecology can communicate with the Android device user in a quick and easy way as many modern mobile devices have a rich set of hardware as many wireless interfaces and good and big screens. A mobile device such as a modern mobile phone is also something that many humans are familiar with and are accustomed to have close at hand. This makes the interaction between a user of a mobile device and a PEIS-ecology more straightforward as the mobile device will not be a new human-machine interface. The graphical interface of a modern mobile phone could very well be used as a good interface point between a human and a smart-home (PEIS-ecology). The mobile devices with Android in a PEIS-ecology can also communicate amongst each other, this adds a new way for communication for Android devices.

With the results of this work it is possible to communicate with PEIS-components and survey a PEIS-ecology by publishing to and retrieving tuples from a tuplespace in a PEIS-ecology. This means that all implemented PEIS-devices can be accessed by an Android device.

The results of this work first of all help the Android application developers that want to add PEIS-functionality to their design, as this now is possible. Second of all this work has resulted in a Java package that most likely outperforms PEIS-JavaJNI when it comes to execution speed for PEIS-components that executes on computers. So this work is also helpful for the PEIS-component developers that develop Java components for computers.

5.2 Future work

For future work the very first thing is the functionality for callbacks. PEIS-Android can be expanded into much, much more. The focus of this work has been on the PEIS API for Android and the Android application part was not explored in any greater extent. An Android application can go through a number of stages from start to end (if any) such as being paused or restarted. What is needed is to test how to best use the PEIS-functionality and the PEIS-kernel in these stages. On a wider perspective the code for using the PEIS-functionality on an Android device can be a very good tool for many purposes. An Android device that has access to PEIS-functionality can access a PEIS-network which can consist of a variety of devices. It could be used for sending data over a
5.2. FUTURE WORK

PEIS-network that can use Bluetooth and WiFi. A speech/messaging communication application could easily be developed and used in areas where the normal communication system malfunctions, or just to skip the fee of using the GSM phone network. Another idea that has come up during development is to be able to send serialized objects. Specifically, whereas so far tuples have been used to share data over an ecology of devices, the tuplespace could be leveraged to share *serialized PEIS-components* – i.e., to allow services to dynamically migrate to different computational platforms. This functionality would not be hard to implement and would open up possibilities for distributed computing and artificial intelligence that were distributed and not bound to a specific device as an example. A tool that would have been developed further if more time were available, was a Android application to view all PEIS-devices and tuples in a PEIS-network. A computer version (*TupleView*) already exists and it would be of great help to those using PEIS-networks.

PEIS-Android only contains an example of how it can be extended. The possibilities for PEIS-Android are many: Access to the GPS for localization of a PEIS-device, using the gyro and the touchscreen for controlling other devices or even for fall detection as in the iFall application [*Section 2.5*] for fall detection. Other extensions could use the camera in combination with image processing for object detection or the microphone to get a PEIS-component that is somewhat aware of the physical environment it is used in.
References


REFERENCES


Appendix A
Getting started

This appendix contains information on how to get started with application
development of Android PEIS-components. I am assuming that a Ubuntu Linux
system is used for development but most points can be adapted to work on
other systems as well. This will not be a complete guide to every aspect of the
installation and usage for a more complete see Androids website [2] as a good
start.

A.1 Getting-Started: Android application
development

The topics described are:

• What is needed
• How to install
• Create a PEIS-component for Android
• Debugging
• Installing and running

A.1.1 What is needed

First there are some programs and software to download:

Eclipse: A good programming IDE that can be extended with plug-ins. The
program can be downloaded from [9] or as in Ubuntu use synaptic or
equivalent.

Android SDK: The Android Software Development Kit that can be found at
Googles Android developers website [2].
Android NDK: The Android Native Development Kit that is very useful if the project that is developed use C/C++-code.

Eclipse ADT plugin: The Android Development Tools is a plugin for Eclipse that makes it easier to develop Android applications with Eclipse.

A.1.2 Installation

Eclipse: The easiest way is to use Ubuntu’s Synaptic to install it.

Android SDK: Unpack the compressed files and place them at a good location and note where the files are placed, because they will be referenced later on.

Android NDK: Unpack the compressed files and place them at a good location and note where the files are placed, because they will be referenced later. Note that the NDK is only needed if the application that is developed access C or C++ code.

ADT plug-in:

- Start Eclipse
- Go to window menu Help -> Install New Software
- Click add
- Enter ADT Plugin for the name
- Enter for the URL location: https://dl-ssl.google.com/android/eclipse/
- Click OK
- Select the Developer tools and click next
- Click Finish and then restart Eclipse
- Click window menu Window and chose Preferences
- Chose if to provide usage statistics to Google if the question come up
- To locate the SDK write it in or click Browse
- Click Apply and then OK
Download SDK components: In Eclipse, go to window menu Window -> Android SDK and AVD Manager. As a minimum install the SDK Tools, SDK Platform-tools and at least one SDK Platform.

Create a virtual android device (AVD): This is the device that will be emulated when testing.

- In Eclipse, chose window menu Window -> Android SDK and AVD Manager. It is also possible to access Android SDK and AVD Manager by clicking the icon in Eclipse that look like the Android logo with an arrow pointing down.
- Click on Virtual Devices.
- Click new.
- Chose the platform that the component/application is aimed for.
- Give the AVD a name and change the settings for it if needed.
- Click Create AVD.

A.1.3 Create a PEIS-component for Android

The following assumes that a new project is created (but it is of course easier to use the getTuple example application project as base right away).

- In Eclipse chose File -> New -> Project.
- Select Android Project. Then write in project details.
- The package name is named as Java packages as example.project1.
- The activity is the main class in the created application.
- The SDK version roughly refer to the Android platform versions where Android 2.2 has API level 8 and Android 3.1 has API level 12.
- After all is filled in click Finish.

The life cycle of the application go through a number of stages that all have functions connected to them. An example of this is the onCreate() that is run when the application start and the onDestroy() that runs when the application is about to be destroyed. Take a look at (if possible) the code for the example application getTuple.java for an example of how to write an application for Android. The application getTuple also give a quick way to see how the file AndroidManifest.xml can be written. The AndroidManifest file contain project specific settings as which permissions the application will have. Another file to look at is the main.xml file which handle the screen interface for the application.
To use the PEIS-kernel code in an Android application create the directory jni in the project directory and place the PEIS-kernel directory there. Create a file called Android.mk in the jni directory. Take a look at (if possible) the example for an Android.mk file in the example application getTuple or try to compose one from this example:

```makefile
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := libpeiskernelandroid
LOCAL_C_INCLUDES := $(call my-dir)/peiskernel
LOCAL_SRC_FILES := \
    core_a_PeisJavaMT.c \
    peiskernel/tuplesAPI.c \
    peiskernel/hashtable.c \
    peiskernel/peiskernel_tcpip.c \
    peiskernel/tuples.c \
    peiskernel/p2p.c \
    peiskernel/peiskernel.c \
    peiskernel/services.c \
    peiskernel/linklayer.c \
    peiskernel/udp.c \
    peiskernel/bluetooth.c
LOCAL_STATIC_LIBRARIES := libpeiskernel
LOCAL_LDLIBS := -lz

include $(BUILD_SHARED_LIBRARY)
```

Included with the NDK are a few example applications that use native code on Android, it can be a good idea to see how those work. Finally include PEIS-JavaJNI and PEIS-Android in your application by import PEISJavaJNI and import PEISAndroid, and that is needed to be able to use the PEIS-kernel functionality on an Android device.

### A.1.4 Debugging

First the project needs to be made into a mixed C/C++ and Java project. The steps for this can be found at:

http://mhandroid.wordpress.com/
/2011/01/23/using-eclipse-for-android-cc-development/

Then quite many number of steps are needed to be able to debug the C/C++ part of the application, which can be found at:
Often the debug environment can not find the bounds of the native functions, the solution for this is to chose Resume (key F8) and the execution will jump back to the Java code. To make jumps easy between Java and native code it is many times necessary to put breakpoints in both the native functions and the Java code. This is so that Resume (key F8) can be pressed and execution can go forward to the next breakpoint. When debugging a project with native code the first step is to start the Java debugging and then start the ADB (Android Debug Bridge). The start of the ADB can be done like this:

```
/usr/local/android-ndk-r5/ndk-gdb-eclipse
--adb=/usr/local/android-sdk-linux_86/platform-tools/adb
--project=/workspace/getTuple/ --force
```

In this example the NDK and the SDK is installed in /usr/local/. The ADB is a tool to handle emulated devices and real Android devices. As an example it can be used to get information (the getprop command) from a device, copy files to and from a device and get the state of a device. To be able to debug on a real device following the steps at:

```
http://developer.android.com/guide/developing/device.html
```

### A.1.5 Installing and running

In Eclipse this is simple as it is just to either open Android SDK and AVD Manager and chose the virtual device to start and then chose window menu option Run -> Run, (or Ctrl+F11) or click the icon that look like a green circle with a white triangle. If the previously mentioned icon is pressed, a virtual device will be started and loaded with the application that is then started. When the emulator is started with the Android SDK and AVD Manager the size of the emulated device can be set which is very useful when there are many different windows on the computer screen.

### A.2 Getting-Started: Using PEIS-JavaJNI

The topics described are:

- PEIS-functionality basics
- How to include PEIS-functionality
- Notes
A.2.1 PEIS-functionality basics

This description of the PEIS-functionality contains the basics needed to get a grasp of what tasks can be performed and how, for a PEIS-component. For a more detailed description of the PEIS-ecology idea and the PEIS-functionality search for PEIS on the AASS web page [1].

**PEIS** stands for Physically Embedded Intelligent System. A **PEIS-devices** is any device that can provide some useful functionality inside a **PEIS-ecology**, that consist of PEIS-devices. **PEIS-components** are what makes the devices into PEIS-devices, they run on PEIS-devices as programs. Each PEIS-component can publish its data to the PEIS-ecology and lend its functionality to other PEIS-components. The functionality in the PEIS-component is accessed by sending a message (with a command) that the component will detect. By **publishing data**, other PEIS-components can use the data from the publishing PEIS-component.

For example a component on **deviceA** publish the temperature at a location and has the functionality to turn on a lamp if it detects a message containing the text ON. Another component on **deviceB** can read the temperature messages and makes the decision to turn on the lamp because it is cold and maybe dark. So it sends out a message containing the text ON, which the component on deviceA will pick up and in doing so turn on the lamp.

All data and commands in a PEIS-ecology are transferred with something called a **tuple**. A tuple is a post in a **distributed database** that every PEIS-component can read. Each tuple has an **owner**, **name** (also called key) and **data**. The distributed database is called a **tuplespace**. If for example a PEIS-component needs access to a tuple containing temperature readings from a specific PEIS-component, it **subscribes** to that tuple. When a PEIS-component subscribes to a tuple it will get access to that tuple.

All PEIS-components use the **PEIS-kernel**, which is the underlying engine for a PEIS-component. The PEIS-kernel is what makes everything work. It handles communication with other PEIS-components, and update of the tuples.

A.2.2 How to include PEIS-functionality

First of all the package PEIS-JavaJNI need to be accessible for your code. If this is true the command `import PEISJavaJNI.*` includes everything you need into the application.

**Class description**: Only a few classes that are needed to get started, these classes are:
• **PeisJavaMT**: Contains the basic functionality that is tied to the PEIS-kernel. It contains functionality to initialize, start, step and stop the PEIS-kernel.

• **Tuple**: The base type for the tuple types in PEIS-JavaJNI. It contains functionality to create, delete, and work with tuples. It contains functionality to publishing and retrieving tuples from a tuplespace as well as setting and getting the fields for a tuple. Objects of this class is usually not created, instead use the specialized types of tuples as the PeisTuple, the AbstractTuple or the MetaTuple.

• **PeisTuple**: Represents an ordinary tuple that is used to transport data and information. It has inherited all the functionality from the class Tuple. This tuple type is what is used most of the time in a PEIS-component.

• **AbstractTuple**: Represents a tuple type that is used as a template. It is not used to transport data in the tuplespace.

• **MetaTuple**: Represents a tuple type that allows to work indirectly with tuples. It do not carry data in a tuplespace but rather it points to another tuple. It can be used for configuration of a PEIS-network as a PEIS-component can know about the MetaTuple which in turn can be set to point to a specific tuple of interest. When doing this the PEIS-component will actually work with the tuple the MetaTuple is pointing to.

• **Subscription**: Represents a subscription. It has functionality to subscribe to a tuple and to get the tuples that match the subscription.

The following description of which functions that can be used and how is for basic usage, there are many other functions available.

**Subscribe to tuples**: In PEIS-JavaJNI this can be done like this:

```java
Subscription sub_atuple1 = new Subscription(-1, "*");
```

The -1 is the owner to subscribe to, a -1 stands for a subscription to tuples with any owner. This is otherwise set to the specific number of the PEIS-component. The * is for the name of the tuple, a * is for a subscription to a tuple with any name. The name can be divided into seven (7) levels with dots (.) like this deviceA.lamp, this would be a tuple named deviceA.lamp. To have levels in the name can be useful if a device publish many tuples but only some are of interest. For example a PEIS-component/PEIS-device might have access to two (2) lamps, each one in a separate room. A subscription to deviceA.room1.lamp gives access to the lamp in room1 that is controlled by deviceA.

**Publish tuples**: To publish a tuple with a text string as data can be done like this:
PeisTuple.setStringTuple("tuple_name", "tuple_message");

*PeisTuple* in the previous example is the class that describes an ordinary tuple for transporting data and information in a tuplespace.

**Retrieval of Tuples:** To get/retrieve all tuples from the tuplespace/PEIS-ecology that match a subscription can be done like this:

```java
Tuple[] results = sub_atuple1.getTuples();
```

If there are any tuples that *match the subscription* (sub_atuple1) these are put into the array of tuples called results.

To get the *owner, name/key and data* from the first tuple in the array called results, can be done like this:

```java
int owner_var = results[0].getOwner();
String name_var = results[0].getKey();
String data_var = results[0].getDataAsString();
```

The following section will explain some of the basics for using the PEIS-functionality in an Android application. **Layouts** are used to set the look of the user interface. The layout is defined in a XML-file in the directory `res/layout` for a Android application project in Eclipse. An example of such a file is this (called tuple-view.xml):

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
    <TextView
        android:id="@+id/tv"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=""
    />
</LinearLayout>
```

The layout is defined in a XML-file in the directory `res/layout` for a Android application project in Eclipse. An example of such a file is this (called tuple-view.xml):
A.2. GETTING-STARTED: USING PEIS-JAVAJNI

android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"
android:text="Get Tuples"
/>
<Button android:id="@+id/publish1"
android:onClick="onPublish1Click"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"
android:text="Publish Tuples"
/>
</TableRow>
</TableLayout>
<RelativeLayout>
    <TextView
        android:id="@+id/tv1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Tuplespace not checked yet"
    />
</RelativeLayout>

A **TextView** is a field for displaying text and the **id** that is set to **tv1** is the name of the component. A **TextView** component is used, in an Android application, like this:

```java
import android.widget.TextView;
private TextView tv1;
    tv1 = (TextView) findViewById(R.id.tv1);
    tv1.setText("Text in textview");
```

The line **android:text=""** for the **TextView** in the XML-file is what will show up in the textview before anything is written to it.

The **button** in the XML-file is a button in the user interface and it can be **linked to a function**, to run when the button is pressed. The link to a function in the Android application is done with the line **android:onClick="function_name"**. The function in the Android application could look like this:

```java
public void function_name(View view) {
}
```

The layout can be set in the **onCreate()** method of the Android application like this:

```java
setContentView(R.layout.tupleview);
```
The onCreate() method is run when the application starts so it is quite natural to put things as setting up the program environment and initializations in this method. To initialize the PEIS-kernel can be done like this (preferably in the onCreate() method):

```java
String[] args = new String[4];
args[0] = "--peis-name";
args[1] = "tupleView";
args[2] = "--peis-hostname";
args[3] = "androidDevice";

try { PeisJavaMT.pj_initialize(args); }
catch (Exception e) {};
```

The array of strings called args contain a few parameters to be used when initializing the PEIS-kernel. In this case the PEIS-kernel/PEIS-component is set to have the name tupleView and that it runs on the host called androidDevice. The other parameters that can be used can be seen in the documentation for the code. Specifically the function `pj_initialize(String[] args)` in the class PeisJavaMT.

### A.2.3 Notes

It does not work to use certain PEIS-functions directly after the PEIS-kernel has been initialize. This goes for subscription so to make a subscription, place it in a separate function and use something like this:

```java
if(sub_atuple1 == null) {
    sub_atuple1 = new Subscription(-1, "*");
}
```

The subscription object sub_atuple1 is defined elsewhere in the main class of the Android application. To make a TextView field scrollable do this:

```java
import android.text.method.ScrollingMovementMethod;

String manyLines = new String("...");
tv1.setMovementMethod(new ScrollingMovementMethod());
tv1.setText(tmpStr);
```

To show a small information message that a button has been pressed do this:

```java
Toast.makeText(this, "Message", Toast.LENGTH_SHORT).show();
```