NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

# INTEGRATION OF INTEROPERABLE ANDROID-BASED COMMAND AND CONTROL SYSTEMS TO CREATE MORE REALISTIC TACTICAL TRAINING

by

Bernd Weissenberger

June 2021

| | |
|---|---|
| Thesis Advisor: | Imre L. Balogh |
| Co-Advisors: | Kirk A. Stork |
| | Christian R. Fitzpatrick |

**Research for this thesis was performed at the MOVES Institute.**

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | *Form Approved OMB No. 0704-0188* |

| | | |
|---|---|---|
| **1. AGENCY USE ONLY** ***(Leave blank)*** | **2. REPORT DATE** June 2021 | **3. REPORT TYPE AND DATES COVERED** Master's thesis |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** INTEGRATION OF INTEROPERABLE ANDROID-BASED COMMAND AND CONTROL SYSTEMS TO CREATE MORE REALISTIC TACTICAL TRAINING | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)** Bernd Weissenberger | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Naval Postgraduate School Monterey, CA 93943-5000 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(E**S) N/A | **10. SPONSORING / MONITORING AGENCY REPORT NUMBER** |

**11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| | |
|---|---|
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** Approved for public release. Distribution is unlimited. | **12b. DISTRIBUTION CODE** A |

**13. ABSTRACT (maximum 200 words)**

This thesis focuses on the interoperability of Android mobile devices during live military training to model the dynamic nature of adversarial forces and enhance realism. The research explores the efficient and effective application of existing interoperability protocols and architectures to transfer and display tactical data to assist ground forces in achieving their training objectives.

Specifically, to address some of the limitations of current training systems that do not support customization, a prototype of an application for Android devices is developed and tested. Consisting of a Mobile Entity Simulator and a Mobile Hit Actor (MHA), the developed prototype proved capable of allowing the devices to connect to a military communication system via Wi-Fi. Once connected, they could send packets to a command and control (C2) system using the distributed interactive simulation (DIS) protocol. Thus, the mobile device could mimic the presence of an arbitrary military unit at the device's coordinates. Moreover, the MHA not only proved successful in registering detonation data but also played a corresponding sound to enhance realism.

To prove the concrete benefits of the application will require further work. For this purpose, experiments should compare the results of live training with and without the use of the developed tools. In addition, the DIS protocol was used in this work; hence, future work should use the High Level Architecture for comparison.

| | |
|---|---|
| **14. SUBJECT TERMS** distributed interactive simulation, high level architecture, modeling and simulation, Kotlin, Android, wireless communication systems, app | **15. NUMBER OF PAGES** 125 |
| | **16. PRICE CODE** |

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT** Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified | **20. LIMITATION OF ABSTRACT** UU |

THIS PAGE INTENTIONALLY LEFT BLANK

# INTEGRATION OF INTEROPERABLE ANDROID-BASED COMMAND AND CONTROL SYSTEMS TO CREATE MORE REALISTIC TACTICAL TRAINING

Bernd Weissenberger
Lieutenant Colonel, German Army
Dipl. Inf. (FH), Hannover University of Applied Science and Arts, 2005

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL**
**June 2021**

Approved by: Imre L. Balogh
Advisor

Kirk A. Stork
Co-Advisor

Christian R. Fitzpatrick
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

This thesis focuses on the interoperability of Android mobile devices during live military training to model the dynamic nature of adversarial forces and enhance realism. The research explores the efficient and effective application of existing interoperability protocols and architectures to transfer and display tactical data to assist ground forces in achieving their training objectives.

Specifically, to address some of the limitations of current training systems that do not support customization, a prototype of an application for Android devices is developed and tested. Consisting of a Mobile Entity Simulator and a Mobile Hit Actor (MHA), the developed prototype proved capable of allowing the devices to connect to a military communication system via Wi-Fi. Once connected, they could send packets to a command and control (C2) system using the distributed interactive simulation (DIS) protocol. Thus, the mobile device could mimic the presence of an arbitrary military unit at the device's coordinates. Moreover, the MHA not only proved successful in registering detonation data but also played a corresponding sound to enhance realism.

To prove the concrete benefits of the application will require further work. For this purpose, experiments should compare the results of live training with and without the use of the developed tools. In addition, the DIS protocol was used in this work; hence, future work should use the High Level Architecture for comparison.

THIS PAGE INTENTIONALLY LEFT BLANK

## TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| 5G | fifth generation (mobile standard) |
| AFRL | Air Force Research Laboratory |
| AGDUS | Ausbildungsgeraet Duellsimulator (training device dual simulator) |
| ATAK | Android team awareness kit |
| AVM | Audio Visuelles Marketing (German company name) |
| C2 | command and control |
| CIV | civilian |
| DHCP | dynamic host configuration protocol |
| DIS | distributed interactive simulation |
| ESPDU | entity state protocol data unit |
| FISH | Fuehrungsinformationssystem Heer (C2 System Army) |
| FRG | Federal Republic of Germany |
| GOV | government |
| GPS | global positioning system |
| HDMI | high-definition multimedia interface |
| HLA | high level architecture |
| IEEE | Institute of Electrical and Electronics Engineers |
| IP | internet protocol |
| ISO | International Standardization Organization |
| JAR | Java Archive |
| JDK | Java development kit |
| JRE | Java runtime environment |
| JS | Java script |
| JVM | Java virtual machine |
| KILSWITCH | kinetic integrated low-cost software integrated tactical combat handheld |
| LVC | live virtual constructive |
| MES | mobile entity simulator |
| MIL | military |
| MIMO | multiple-in multiple-out |

| | |
|---|---|
| MHA | mobile hit actor |
| NATO | North Atlantic Treaty Organization |
| NFC | near field communication |
| NGA | National Geospatial Intelligence Agency |
| OS | operating system |
| P2P | peer-to-peer |
| PDU | protocol data unit |
| SSH | secured shell |
| USA | United States of America |
| USB | universal serial bus |
| VM | virtual machine |
| VPN | virtual private network |
| VR | virtual reality |

# I. INTRODUCTION

Computer-aided training has become an integral part of everyday military life. It saves time, money, and can improve training by making it more dynamic and realistic. This is especially true for live military training events at the battalion level and higher as computers are integrated across the units within the North Atlantic Treaty Organization (NATO) armed forces. In particular, the Command and Control (C2) systems integrated in tactical vehicles (trucks, tanks, troop transporters, and all others) are indispensable in ensuring blue force tracking and keeping a tactical situation map at the same time—during training as well as in the real world. As this master's thesis is being written, the combat units of the German Armed Forces are using the C2 system "FISH (Fuehrungsinformationssystem Heer—Command and Control System Army)," which is described in the following section. Undoubtedly, efficient training using integrated systems is essential for the successful deployment of a nation's army. Furthermore, such a practice reinforces the Latin proverb "exerce te talem, qualem pugnas" (train as you fight), which has not lost any of its validity even today.

## A. OVERVIEW

In the German Armed Forces, we have training centers called "Gefechtsuebungszentren" (Combat Training Centers). In these centers, our military trains in a combination of "Live," "Virtual," and "Constructive" environments. This allows a battalion to train in this facility with just three real companies at the training ground. The other three companies are virtual only.

The vehicles within these scenarios are equipped with (at least) two different systems: AGDUS and FueInfoSys.

(1) AGDUS "Ausbildungsgeraet Duellsimulator" (Training Device Duel Simulator)

AGDUS, which is a transmitter adapted for the weapon of a particular vehicle, emits a coded laser beam when a shot is fired. Various data such as the shooter's identity number (vehicle), type of weapon and ammunition are transmitted to enable the identification of who has hit whom with which weapon. If the receiver is hit optically, an acoustic signal is emitted by the control unit. Depending on the hit zone, a built-in display shows the damage, from light to medium to severe damage and the associated downtime, as well as a complete failure. The decision about the severity of the damage depends on the results of engagement.

(2) FueInfoSys "Fuehrungs Informations System" (Command and Control System, C2 System)

The FueInfoSys displays the current position of a vehicle, a (sub) unit, or an entire battle group in near real time on a digital map. This situation report is available to the commander as well as on any other vehicle, broken out to the appropriate unit level.

## B. CURRENT CAPABILITIES AND LIMITATIONS

The main problem with the current system configuration just described is that both the units and vehicles of the unit being trained, and the vehicles of the opposing force are displayed with their actual vehicle signature (tactical symbol) in the C2 system. Since the Bundeswehr cannot always provide sufficient combat vehicles and for every exercise, trucks are sometimes used to play the role of target systems. For this purpose, they are equipped with the appropriate AGDUS sensors and can thus react to simulated fire. For example, a battle between two tank battalions is practiced "blue against red." The blue training force is equipped with actual the Leopard 2A6 main battle tank. The opposing side is said to be an enemy tank battalion equipped with T80s, which are simulated by trucks from a blue unit. In the C2 system of the exercise, however, the enemy "tanks" are represented as blue trucks and not as red battle tanks. This not only leads to confusion with the tank crews but also with the trained battalion staff in the command post, who is shown the same operational picture. Since the transmission units for vehicle recognition cannot easily be changed or adjusted, this deficiency is often accepted, and training is conducted with this artificiality.

Another limitation is the lack of a virtual hit display in the real world. While a hit with the AGDUS laser is indicated in the C2 system, the gunner and the driver of the vehicle do not see whether the enemy has been successfully engaged, and consequently, they react differently in this case than they would otherwise. If a hit were also recognizable in reality, the entire tank crew could turn its focus on a new target to interdict.

## C. PROBLEM STATEMENT

As previously described, the representation of a combat vehicle in a C2 system during live training cannot be changed (Figure 1). German C2 systems such as the Army Command and Control System or the Command and Control System for Army (FISH) do not allow this customization for good reason, as this data needs to be accurately displayed to avoid friendly fire (blue force tracking). What is vital in combat and real operational conditions, however, achieves exactly the opposite with live training; namely when, as described earlier, our own trucks are used as target representations for troops. The currently used systems do not allow the vehicle signatures to be changed. The work in this thesis investigates the feasibility of developing a software solution to address this deficiency and whether a similar approach could also be used to create a virtual hit display device.

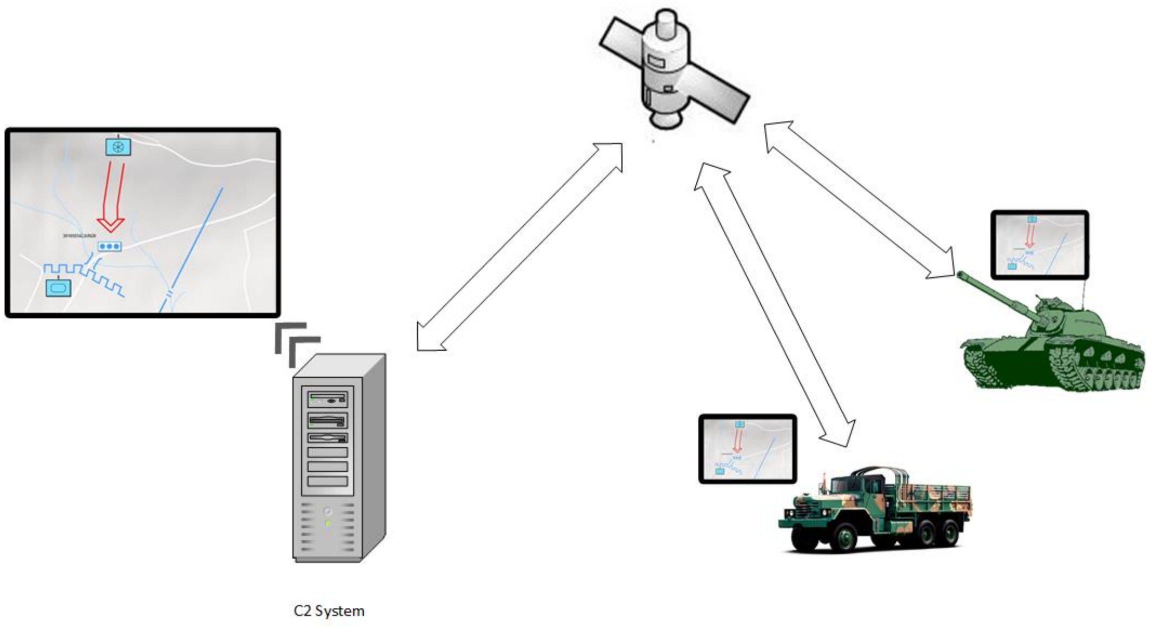


Figure 1. Live Training Actual Situation

My proposed solution is to create a system to augment the current C2 environment to address the problem just described. This research involves building an application for commercial Android devices on which an operator can select the platform being proxied and then use the mobile device to classify the platform as required to augment the tactical scenario. As this solution does not exist today, there are a few research questions to address as the proposed solution is built.

1. What wireless communications systems are available to network Android devices, and what are the advantages and disadvantages of each for the proposed application?
2. What simulation interoperability protocol or architecture should be used to pass this tactical information and why i.e., what potential architectures need to be researched)?
3. Are there significant differences in operating the Distributed Interactive Simulation (DIS) protocol versus the High Level Architecture (HLA) when a number of devices are all streaming to an existing simulation system for visualization?
4. Once this data is broadcast as desired, how will it be received and bridged into the tactical system used to integrate into the training environment?
5. Is it possible to develop a prototype on a mobile (Android) device that considers and confirms the previous results? What programming language should be used and why?

## D. MOTIVATING CONSTRAINTS

The motivation for this work is to address a deficiency in how information is displayed during training exercises in Germany. Ideally, the development of a solution should be done with the actual systems used; however, for various logistic and other reasons access to these systems was not available for this work. Therefore, this work focuses on developing a system to investigate the feasibility of the proposed approach. To ensure that the results obtained in the thesis apply to the actual systems being used for training, the goal is to use a development/test system configuration that is a close match to the actual fielded systems.

The feasibility of improving the live training conditions using additional systems is examined. The goal is to determine whether the signature of a vehicle can be changed by adding a self-developed system and whether it can be configured to behave as any possible target system. This is demonstrated by having a mobile device send a certain signature (military symbol) in a protocol understood by the C2 system and having the same symbol displayed in the situation report. The reception of the data and the display of the symbol should take place immediately without a noticeable delay (< 1s).

In a second step, the feasibility of showing the effects of virtual events using effectors is demonstrated. This is demonstrated by having a mobile device send a signal to a specific vehicle also. The reception of the data and the resulting reaction should take place immediately without a noticeable delay (<1 s).

## E. ASSUMPTIONS

The goal of this work is to develop a model system that can potentially transfer to the real system used for training. For this reason, the assumption is that the system used to develop this model is similar enough to the systems used in Germany to facilitate such a migration of the software. The components used in this research project are described in detail in the following sections.

### 1. Hardware

Modern combat vehicles and military trucks normally contain modern communication systems (ComSys) that enable IP-based, encrypted data transmission via various channels such as HF radio or satellite communication (Figure 2). In addition, these ComSys offer a range of connection options for external devices. These connections can be wired (Ethernet, USB) or wireless (Bluetooth, Wi-Fi).

Source: www.thalesgroup.com

Figure 2. Example of a ComSys.

In the present work, a freely configurable router from the German company AVM is used to replicate the ComSys for development purposes. The encrypted data transmission is displayed and simulated using a virtual private network (VPN).

## 2. Software

The transmission of location data on the real-world vehicles and other information is based on the distributed interactive simulation (DIS) protocol. Both the transfer from the vehicle to the main computer and vice versa take place using this protocol. The transmission of one’s own military symbol with the respective location at a certain time cannot be changed, but it can be suppressed or deactivated.

Instead of a real C2 system, the VR-Forces software from MAK Technologies is be used. This can take on the role of both the required protocol (DIS) processing and the representation of a situation map-like scenario.

# II. RELATED WORK AND EXISTING TECHNOLOGIES

In this chapter, existing technologies and the feasibility of their use in addressing the stated problem are investigated. Specifically, the chapter presents a comparison of two major software solutions, the most common communication technologies as well as different protocols working on C2 systems.

## A. AVAILABLE SOFTWARE SOLUTIONS

As a first attempt to solve the described problem of incorrectly displaying enemy vehicles for target display, this research compares and evaluates existing software products that could represent a solution.

### 1. Android Team Awareness Kit

Android Team Awareness Kit (ATAK) is an application developed for Android mobile devices. Its purpose is to offer map data and military situation awareness. Furthermore, it allows navigation, targeting, and data sharing. The Global Positioning System (GPS) provides the application with the needed geospatial data. It uses the military symbol standard MIL-STD-2525B to display units and objects, but the application is also able to use customized icons from Google Maps or Google Earth. Developed in 2010 by the Air Force Research Laboratory (AFRL), this application had minimal acceptance at first. Since 2016, however, an increasing number of users adopted this tool, and in 2020, about 250,000 civilian and military users were working with it (*Android Team Awareness Kit*, 2021).

The first version, developed for Android 2.1, had problems with Java Archive (JAR) files in runtime. Therefore, the developers looked for a solution that makes it unnecessary to compile the entire codebase when adding additional functions. They came up with the idea of using a plug-in framework based on Java Script (JS). Normally, JS is not able to access the network or file resources. AFRL solved this by allowing only JS files in a specific folder on the mobile device these rights (Carpenter & Carpenter, 2013).

Today, there are five versions of ATAK available:

- ATAK—Public Release (ATAK-PR): as the name says, it is for public individuals and public use.

- ATAK—Civilian (ATAK-CIV): this version is mainly used by first responders.

- ATAK—Government (ATAK-GOV): this is a restricted version for U.S. Government use only.

- ATAK—Military (ATAK-MIL): as the name implies, this version is used only by the U.S. Military and foreign military (when provided by the United States). It is not available for private use.

- ATAK—Five Eyes (ATAK-FVEY): this version is used by intelligence agencies of Australia, Canada, New Zealand, the UK and the United States.

In a recent white paper, Jeff Henderson, a strategic account manager for Army and special services at Panasonic, identifies five essential advantages of ATAK:

1. ATAK is an effective and simple situational awareness (SA) application.
   [Capabilities like blue-force and red-force tracking are intuitive and easy to use. Also, the communication module is very easy to use.]
   Henderson: "It's literally like texting. You pull up the little message app and send a message directly to another teammate or to the whole team".
2. ATAK is a user friendly and easy to stand up app.
   [It is easy to use. Setting up a simple environment will take no longer than 30 minutes.]
   Henderson: "I think that's why it's become so popular. It's a very effective, simple, straightforward tool".
3. ATAK is a flexible, lightweight, and free solution for warfighters.
   [A huge advantage is that because of its design it is flexible enough to run on a mobile device, and integration of new features is very easy.]
   Henderson: "ATAK is so lightweight that you can really put it on any Android device. It definitely gives you a lot more flexibility than most tools".
4. ATAK is evolving to meet growing challenges.
   The development of warfighting never stops. Because of its design, ATAK can deal with all the new challenges. [For example, jamming of

communication systems is growing nowadays. With a third-party tool, ATAK can warn users about this threat.]
Henderson: “You can still do all the simple SA activities, but you can also do all of these other things. The capabilities are really growing”.

5. ATAK is moving beyond the battlefield.
[The fields of use are expanding. Not only military personnel can take advantage of it, but also groups like firefighters, first responders, and homeland security can use it effectively for their purposes.]
Henderson: “If you picture responding to a wildfire or flood, or any fairly chaotic situation where you have to make decisions quickly, that’s a perfect environment to leverage the features of an ATAK application” (Panasonic, 2020).

### 2. Kinetic Integrated Low-Cost Software Integrated Tactical Combat Handheld

Kinetic Integrated Low-Cost Software Integrated Tactical Combat Handheld (KILSWITCH) is application running on an Android device. It is used by the U.S. Marines to provide them air and ground, real time situational awareness (SA). This software uses a map as a background and does not need a connection to a server (Sadler & Metu, 2017).

KILSWITCH was developed by Naval Air Warfare Center Weapons Division, China Lake to support the U.S. Marines coordinating precision air power in 2012. According to an article published in the *MCA Marines Gazette*, the application has three main features (Barksdale, 2014):

1. Rapid display of National Geospatial-Intelligence Agency (NGA) map and imagery data (in organic Department of Defense formats) within a GPS-enabled moving map display. This capability facilitates the exploitation and operational use of NGA geospatial intelligence at the lowest tactical level.
2. Easy generation of precise location and elevation data for any imagery significant feature, to include targets.
3. Ability to seamlessly search and scroll Marine Corps Intelligence Activity–developed compound maps or gridded reference graphics (GRGs) by sheet, sector, and compound. (Barksdale, 2014)

In 2018 several websites (e.g., cyware.com, marines.mil) reported that according to an unnamed whistleblower, the KILSWITCH software contains several vulnerabilities in live combat scenarios.

Further, in December 2018. a report of the March 2017 investigation released by the Office of the Naval Inspector General (OSC DI-17-3391 NAVINSGEN 201702142) was made public. Although the report does not state the nature of such vulnerabilities, it talks about failures in communication. It is important to note that KILSWITCH was never planned for use in live combat scenarios. It was always expected to be used in military exercises only. Therefore, cybersecurity was not a concern for the developers. The main issue was that using the application can enable the enemy to get access to sensitive battlefield information or location data. Notably, for a couple of months the U.S. Marines were able to download the KILSWITCH application to their private devices, which may not provide good security protections.

The report of the Office of Naval Inspector General gives some recommendations and advice on how to deal with this problem. The KILSWITCH software should no longer be used in real battle scenarios and all versions of this application on private devices should be deleted immediately. Instead of using KILSWITCH, the report recommends the using ATAK instead. Following the doctrine "train as you fight," KILSWITCH should also be banned from exercises as well.

Each developer of software should be aware of cyber vulnerabilities. Even though KILSWITCH was designed for use in exercises only, its developers should have taken security into consideration. Furthermore, distribution of military software must be controlled by special military personal. Making software available to all soldiers and installable on their private devices will always generate cyber security issues and is a welcome possibility for foreign intelligence services.

## B. COMPARISON OF AVAILABLE COMMUNICATION SYSTEMS

This section investigates the different ways in which mobile devices could be connected to an existing C2 system. The aim is to get an overview of the state of the art at the time of writing this thesis. My investigation focuses on which connection types exist in common mobile devices and which parameters these connection types have.

### 1. Near Field Communication

In 2002 Near Field Communication (NFC) technology was initiated by Sony and NXP Semiconductors (formerly Philips). They defined a technology specification and created a technical outline. Then in 2003 NFC was approved as an International Organization for Standardization (ISO) standard and a European Computer Manufacturers Association (ECMA) standard.

### *a. Specifications*

This technology is used to transfer data between two devices closer than 10 cm with a data rate of 424 kbps (Hossein Motlagh, 2012). It was initially used for contactless tracking of Radio Frequency Identification (RFID) tags. Today, it is also used for data transfer and contactless payment systems (Ylinen et al., 2009). The frequency used is 13.56 MHz. This frequency is globally available and license free.

### *b. Basic Operation Modes*

This technology is used by two different types of devices: active and passive. The passive devices are RFID tags, which contain a simple type of unique identification (ID). This ID can be read by active devices like smart phones, which can operate in three different modes.

#### (1) NFC Card Emulation Mode

In this mode the mobile device is used like a smart card or RFID tag (passive mode). The data is stored in a virtual chip and can be read by an active device. Main use cases for this mode are payment systems or access authorization systems like Tesla's keyless system or authentication technique at charging stations for electric vehicles. The user must hold the mobile device close to a special reader device. The reader device sends a trigger signal and the smart phone responds with the requested data (credit card data, access code, etc.).

#### (2) NFC Reader/Writer Mode

In this mode the mobile device can be used as a reader or as a writer (active and passive). A special reader device can be used like in card emulation mode—just to read data from a smart phone or the smart phone can be used to read data from an NFC sender device. This mode is often used in museums where visitors can use their smart phones (equipped with a specific app) to get more information about an exhibit by holding the phone close to a tag belonging to the object on display.

(3) NFC Peer-to-Peer (P2P) Mode

This is the most powerful mode. Like in a Wi-Fi network, both devices are connected in an ad-hoc fashion. This mode is used to exchange or transfer data from one device to another. The devices could be of the same type (smart phone to smart phone), or they could be very different devices (car to smart phone, smart phone to car).

#### *c. Summary*

The advantages of NFC are its low power consumption and high availability in all common mobile devices. Its disadvantages are low bandwidth and the very limited working range.

### 2. Bluetooth Technology

In 1998, five companies from Europe, as well as from South Korea, and the United States (Nokia, Erikson, Toshiba, IBM, and Intel) established the Special Interest Group (SIG) to develop a new universal wireless connectivity standard to connect mobile devices. The goal was to generate a license-free technology that could be used by all companies in the world (Bhagwat, 2001). This technology aimed to replace cables wherever possible, be driven by low power, and prevent mutual disturbances.

The result was “Bluetooth,” named after a Danish King Harald Bluetooth, who united the two warring states Denmark and Norway (as an analogy to the connection between two devices produced by different companies).

The first version, version number 1.0, was released in 1999 by SIG, and it had a maximum bandwidth of 732.2 kBits/s. Because of a security problem a new release was necessary in 2001 (version 1.1). Further versions followed at intervals of two to three years, which in addition to increasing the resistance to interference sources also increased the bandwidth. Finally, in 2009 the “Seattle Release” (version 3.0) with a maximum of 480 MBits/s was published. This was the last version with normal power consumption. The range of this version was between 1 mW (class 3) and 100 mW (class 1) (Ferro & Potorti, 2005).

Late in 2009 the first “Low Energy” standard (version 4.0) came out. It was followed by version 4.1 in 2013, 4.2 in 2014, and 5.0 in 2016. The Samsung Galaxy S8 (offered since 2018) was the first smartphone with integrated Bluetooth version 5.0. Today, about 80 percent of all devices in use are equipped with this standard.

The newest specification standard is version 5.1, released on January 28, 2019 (Woolley, 2019). Using that standard makes it possible to determine the direction of a Bluetooth signal transmission. This is achieved by using an array of antennas.

### a. *Specifications of Different Bluetooth Versions*

Table 1 gives a brief overview of the differences between the various versions of Bluetooth.

Table 1. Bluetooth Versions

| | **Bluetooth version** | | | | |
|---|---|---|---|---|---|
| | **1.x** | **2.x** | **3.x** | **4.x** | **5.x** |
| **Speed (Mbit/s)** | ~1 | ~2 | ~24-480 | ~24-480 | ~24-480 |
| **Range (m)** | 100 | 100 | 100 | 10-100 | 10-100 |
| **Low Energy Mode** | no | no | no | yes | yes |
| **Dual-Profile**[1] | no | no | no | yes | yes |
| **IPv6** | no | no | no | no | yes |
| **Pairing with NFC** | no | yes | yes | yes | yes |
| **AES 128Bit encryption** | no | no | no | yes | yes |

Adapted from https://de.wikipedia.org/wiki/Bluetooth

### b. *Basic Operation Modes*

If a Bluetooth device is turned on, it will usually operate as a slave device. It waits for a signal from a master that wants to connect to it (inquiry phase). When the connection

[1] A Bluetooth device could be master and slave at the same time. This is possible since version 4.0.

is established, the devices can exchange data. A master device in this case could have more than one active or parked slave device, which is called a **piconet** configuration (Figure 3).

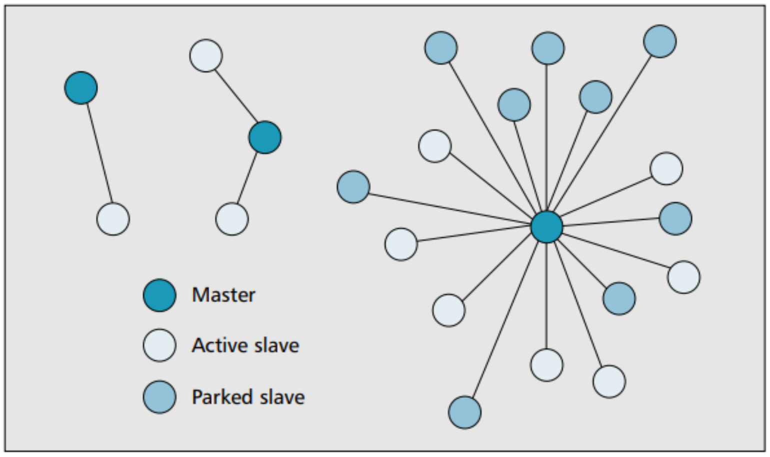


Figure 3. Bluetooth: Piconet Configuration
Source: Ferro and Potorti (2005)

Since version 4.0 a Bluetooth device can operate as master and slave at the same time. This allows for the connection of two or more piconets to a **scatternet** (Figure 4). Now, a device can communicate with other devices in other piconets (multihopping).

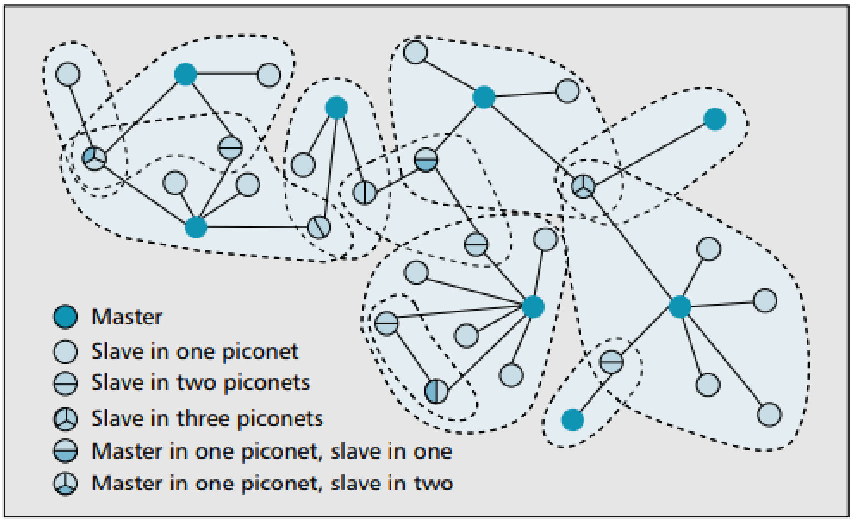

Master
Slave in one piconet
Slave in two piconets
Slave in three piconets
Master in one piconet, slave in one
Master in one piconet, slave in two

Figure 4. Bluetooth: Scatternet Configuration
Source: Ferro and Potorti (2005)

#### c. *Summary*

The advantages of Bluetooth technology are its relatively low power consumption and its widespread availability on most available mobile devices. The disadvantages of this technology are its limited range and lower bandwidth compared with other technologies.

### 3. Wi-Fi Technology

In 1990, the Institute of Electrical and Electronic Engineers (IEEE) started a project with the objective to develop a standard for wireless communication between computers. The basic goal was to define a Medium Access Control (MAC) system and the physical layer (PHY) attributes. The name of that project was IEEE 802.11.

The project work was based on prior proprietary developments by various vendors. In these early efforts two different frequencies were used to transmit data between computers. The first one was 900 MHz, which allowed a data transfer rate of ~1 Mbps. This was ten times slower than the speed provided by wired Local Area Networks (LAN) at the time. The second frequency used was 2.4GHz, which enabled data transfer at the same speed as that offered by wired LANs. The problems with these early wireless network implementations were that they were each proprietary and so were not interoperable; were susceptible to interference, which impacted their reliability, and they were relatively expensive (Bhoyar et al., 2013). The goal of creating the IEEE standard was to address these problems.

In Europe and the United States the spectrum used ranges between 2.4 and 2.4835 GHz while Japan it ranges between 2.471 and 2.497 GHz (Ferro & Potorti, 2005). This band is called the industrial, scientific, and medical (ISM) band. To reduce interference in this band, the total range was divided into 14 channels. In the United States only channels 1–11 are permitted for use. Channels 12 and 13 are for low-power usage only and channel 14 is banned. In Europe channels 1–13 are permitted for use, and in Japan channel 14 is permitted for use. The range at 1000 mW sending power is between 35 m and 300 m.

Today, the most commonly used standards are IEEE 802.11n and IEEE 802.11ac. In addition to 2.4G Hz, these newer standards allow the use of channels in the 5GHz range. This allows for possible link rates up to 1.35 Gbit (Watson, 2012).

### *a. Specifications and Basic Characteristics*

Table 2 gives a brief overview of the differences between various versions of Wi-Fi technology.

Table 2. Wi-Fi Generations and Parameters

| Standard | Adopted | Frequency | Speed | Range (outdoor) |
|---|---|---|---|---|
| **IEEE 802.11** | 1997 | 2.4GHz | 1-2Mbits | ~120m |
| **IEEE 802.11a** | 1999 | 5GHz | ~54Mbits | ~120m |
| **IEEE 802.11b** | 1999 | 2.4GHz | ~11Mbits | ~140m |
| **IEEE 802.11g** | 2003 | 2.4GHz | ~54Mbits | ~140m |
| **IEEE 802.11n** | 2008 | 2.4/5GHz | ~248Mbits | ~250m |
| **IEEE 802.11ac** | 2014 | 5GHz | ~6.77Gbits | ~250m |
| **IEEE 802.11ax** | 2019 | 2.4/5GHz | ~9.6Gbits | ~250m |

Adapted from https://de.wikipedia.org/wiki/Wireless_Local_Area_Network

### *b. Basic Operation Modes*

There are basically three different operating modes employed with Wi-Fi systems. First, there is *ad-hoc* or *Wi-Fi direct* mode, second, *infrastructure* mode, and third, the *multiple access point* or *extended service set* mode.

#### (1) Ad-Hoc Mode

In Ad-Hoc mode each computer is connected directly with each other (Figure 5). No device is between them to manage the traffic. Ad-Hoc mode has been available since version IEEE 802.11a and is most often used by multiplayer handheld game consoles, such as *Nintendo DS* and *PlayStation Portable*.

| | |
|---|---|
| Advantages: | - no additional hardware is needed |
| | - no central station |
| Disadvantages: | - maximum range is limited to the maximum range of one of the devices |
| | - interference (many channels need to be used) |

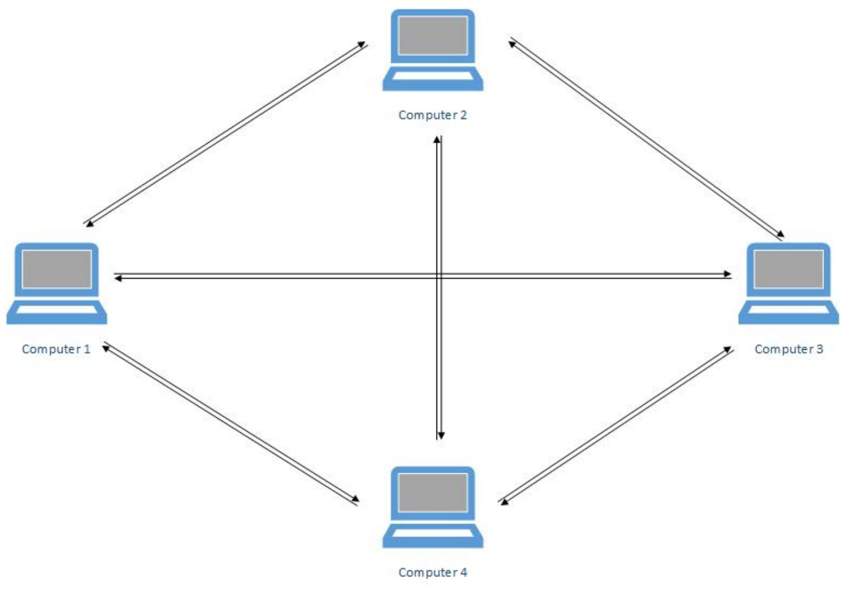


Figure 5. Wi-Fi Ad-Hoc Mode

(2) Infrastructure Mode

Infrastructure mode is the most common mode used. In this mode each computer is connected to an access point (AP) (Figure 6), where the access point manages the connection. Using this mode can extend the range since every computer (or client) that can reach the AP can communicate. In this case, the range is limited by the range of the AP rather than the range of the most limited computer.

| | |
|---|---|
| Advantages: | - easy to manage |
| | - extended range |
| Disadvantages: | - one central point (the AP is a single point of failure) |

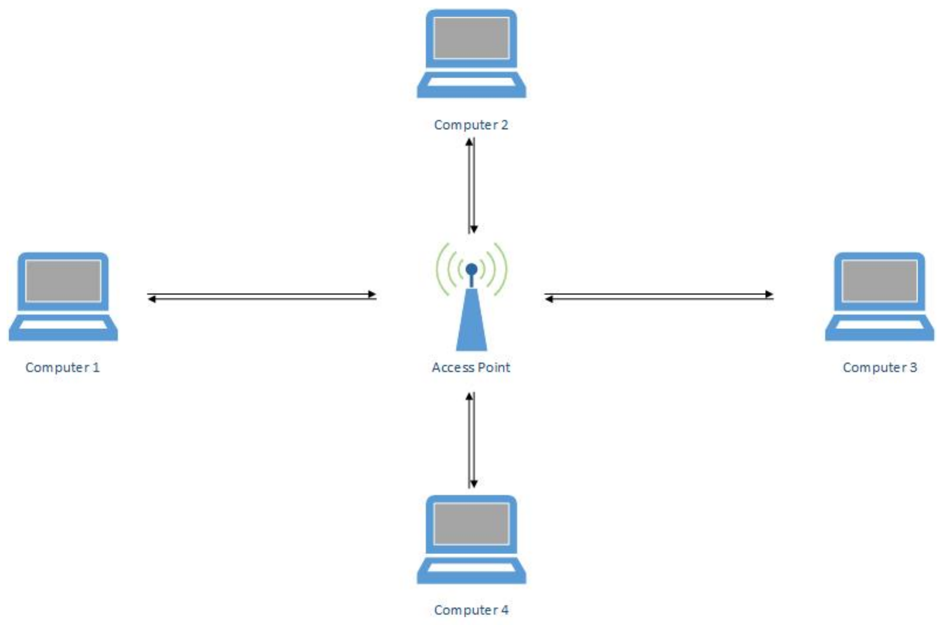


Figure 6. Wi-Fi Infrastructure Mode

(3) Multiple Access Point Mode

Multiple Access Point Mode or Extended Service Set (ESS) is basically a union of several infrastructure mode driven networks. Two or more wireless local area networks (WLAN) are connected by a switch. In this configuration, the networks' ranges must overlap and the service set identifier (SSID) must be the same. As shown in Figure 7, computer 5 is connected with the left WLAN. When computer 5 moves out of the range of the left AP and into the range of the right WLAN, as long as this WLAN has the same SSID, computer 5 connects automatically to the new network and is reachable without interruption.

| | |
|---|---|
| Advantages: | - extended range |
| Disadvantages: | - complex configuration |

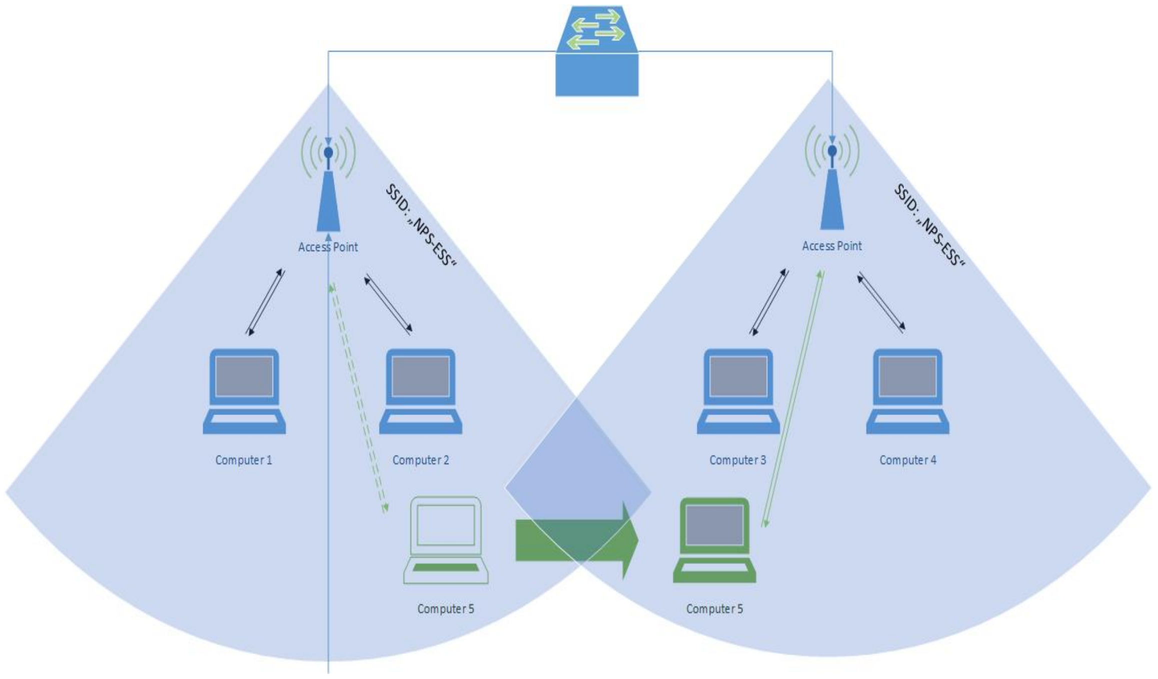


Figure 7. Wi-Fi Extended Service Set

### 4. USB Technology

In 1996 Universal Serial Bus (USB) technology was introduced by the USB Implementers Forum (USB-IF), which was a non-profit organization created to support and promote the USB standard. Founding members were seven companies: Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel (Tailor, 2015). In 1998 the Apple iMac was the first mainstream personal computer equipped with a USB port.

The goal of the USB-IF was to develop and standardize a common interface for connecting personal computers to peripheral devices such as keyboards, mice, cameras, disk drives, and printers. Almost all existing interfaces from the time have been replaced by USB connection today. In addition to being a data transfer protocol, the USB standard also included a power supply component to allow power to be provided to the connected devices through a single connection. The USB Power Delivery standard allows for up to 100 W of power to be supplied to devices.

#### *a. Specifications and Basic Characteristics of Different Versions*

Since 1996 seven versions of the USB standard have been developed and released. The most recent one was USB 4 in 2019. Each version has its own parameters and capabilities. This section compares these versions along with their parameters and capabilities (for an overview see Table 3).

(1) USB 1.0

Defined in 1996, USB 1.0 has a data transfer rate of 1.5 Mbit/s at low speed and 12 Mbit/s on full speed. The lower speed was used for less expensive peripheral devices like the mouse, keyboards, and so on. Since a high transfer rate was not needed, an unshielded cable could be used for such devices. The full speed mode was used for devices like external floppy disks and printers and required shielded cable connectors (*USB*, 2021).

(2) USB 1.1

In 1998 USB 1.1 was released to solve some minor problems of USB 1.0, such as not supporting an Interrupt Out Transfer (IOT). Neither USB 1.0 nor USB 1.1 provided a standard for the connector to be used to connect to the peripherals. All connections were treated as using fixed attached cables (*USB*, 2021).

(3) USB 2.0

In 2000 USB 2.0 was released. It specified a data transfer rate of 480 Mbit/s. Most importantly, it included the definition of small physical connectors (type A and type B), which allowed for the connection of external hard disks and video devices using standard pluggable connections (*USB*, 2021).

(4) USB 3.0

In 2008 USB 3.0 *Super Speed* (SS) was released. This allowed for possible data transfer rates up to 4 Gbit/s. To achieve this, new connectors and cables were used, but they were only partially compatible with previous versions (*USB*, 2021).

(5) USB 3.1

In 2013 USB 3.0 was replaced by USB 3.1 (also called USB 3.1 Gen 1 or *Super Speed+*). The data transfer rate was increased to 10 Gbit/s (*USB*, 2021) in this release.

(6) USB 3.2

In 2017 USB 3.2 enabled a double transfer rate of 20 Gbit/s. Moreover, there was no difference in the protocol compared with USB 3.1. The additional speed was achieved by allowing a second pair of cables to be used in parallel (*USB*, 2021).

(7) USB 4

This latest standard was introduced in 2019. It is the common successor to USB 3.2 and the Thunderbolt 3 standard, which was used by Intel and Apple for high-speed data transfer. Data transfer speeds up to 40 Gbit/s can be attained. This enabled the transmission of 4 k video data with 10 bits per color channel and 60 Hz refresh rate (*USB*, 2021).

Table 3. USB Standards

| **Standard** | **Since** | **Data Transfer Rate** | **Max Power** |
|---|---|---|---|
| **USB 1.0** | 1996 | 12 Mbit/s | 0.5 W |
| **USB 1.1** | 1998 | 12 Mbit/s | 0.5 W |
| **USB 2.0** | 2000 | 480 Mbit/s | 2.5 W |
| **USB 3.0** | 2008 | 4 Gbit/s | 4.5 W |
| **USB 3.1** | 2013 | 10 Gbit/s | 4.5 W |
| **USB 3.2** | 2017 | 20 Gbit/s | 7.5 W |
| **USB 4** | 2019 | 40 Gbit/s | 7.5 W |

Adapted from https://en.wikipedia.org/wiki/USB

Today, the USB standard is widely used. The high data transfer rate supports data-intensive use cases like video streaming and real-time solutions. In addition, the ability to use the data cable for power supply reduces cable clutter.

## 5. Mobile Standard Fifth Generation (5G)

5G is the newest telecommunication standard expected to extend or replace the 4G LTE (Long Term Evolution) network. Among the major innovations introduced by 5G technology are the significantly higher data transfer rate, very low latency, better network coverage through more radio cells, and a significantly improved Quality of Service (QoS) (Noohani & Magsi, 2020).

In December 2018, the 3rd Generation Partnership Project (3GPP), which oversees the 5G standard, launched release 15, the first document that standardizes the capabilities of 5G. The actual release (release 16) was published July 2020.

Today, 5G is still primarily in the planning stage. Relatively few experimental cells exist and client devices that support 5G are not widely in use. 5G will be the base for future data transfer requirements. The Internet of Things (IoT), autonomous cars, and the network connectivity in areas with no (wired) internet connection are important goals that will not be attainable without a powerful digital infrastructure. In addition, existing networks are faced with an increasing number of users of mobile devices with increasing data transfer rates (Nordrum et al., 2017), which is further pushing for the expanded bandwidth that 5G promises.

### *a. Specifications and Basic Characteristics*

Currently, because of the limited availability of fully functioning 5G infrastructure, most performance data are based on the theoretical values of the parameters of 5G specification. The German "Informationszentrum Mobilfunk" (Mobile Ration Information Center) published a 2020 paper "Daten und Fakten zur fuenften Mobilfunkgeneration" (Data and Facts about 5G) reporting the first results from a field study done in Hamburg. They found the actual measured range for 5G was about **150 meters** and a realistic data transfer rate was **10 Gbit/s,** which is 40 times higher compared with 4G LTE (max 225 Mbit/s). Transmission power of base stations in the field study was **10 Watts** as compared to 40 Watts for 4G. The transmission power of the mobile devices was equal to that in 4G—depending on distance (Informationszentrum Mobilfunk, 2020).

### *b. Basic Operation Mode*

Since the effective transmission range for 5G is quite short due to the shorter wavelength used, this is compensated for by using more cells with smaller ranges. By doing this, areas that would otherwise be in the radio shadow can also be reached and covered. To achieve adequate coverage in an urban environment, for instance, one base station is needed every 250 meters (Nordrum et al., 2017). While this increases the number of cells, because of the short wavelength being used, the antennas can be very small and also the transmitting power of the base stations can be lower (about 10 W) compared with 4G base stations. Further, each base station needs to be connected to at least one other base station. Because of this need for a higher density of towers to provide good coverage, however, it will be difficult to supply rural areas with 5G-connectivity.

Using massive multiple-in, multiple-out (MIMO) radio communication enables 5G to handle significantly more client devices per base station when compared to 4G base stations. This, and because 5G will have more base stations, increases the rate of supplied devices by up to a factor of 22. The huge number of very small antennas on each base station connected in an array enables beam forming (see Figure 8), and consequently achieves new records for spectrum efficiency (Nordrum et al., 2017).

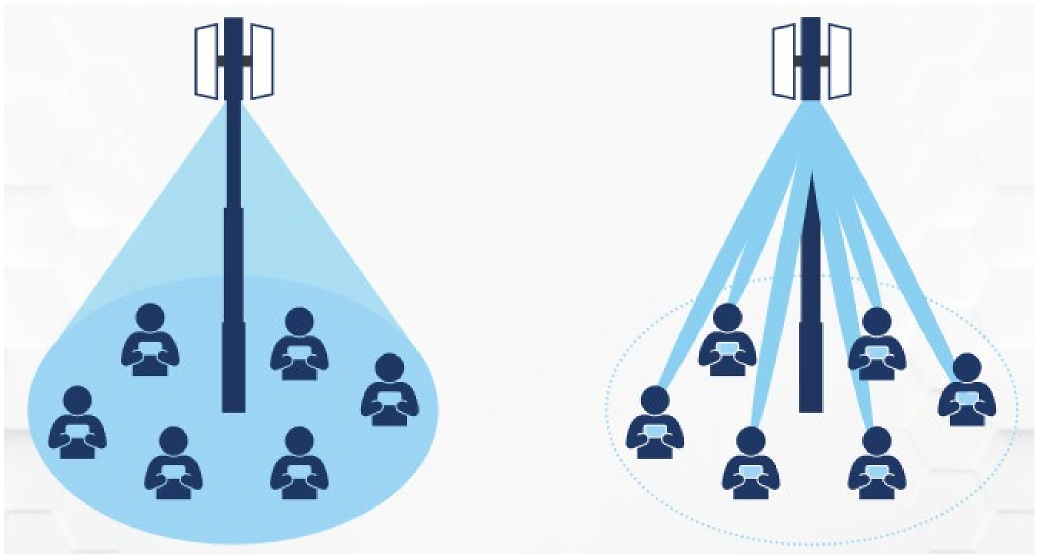


Figure 8. G Beam Forming with Antenna Arrays.
Source: Nordrum et al. (2017).

Today’s base stations and cellphones rely on transceivers that must take turns if transmitting and receiving information over the same frequency, or operate on different frequencies if a user wishes to transmit and receive information at the same time (Nordrum et al., 2017).

## C. AVAILABLE PROTOCOLS AND ARCHITECTURES

In this section we look at the data transfer protocols and architectures that are currently in use to manage this interchange of information between military simulations.

### 1. High Level Architecture

Steffen Straßburger, a professor at Technical University Ilmenau/Germany, describes High Level Architecture (HLA) as an IEEE standard for distributed simulation. The focus of this architecture is the interoperability, reusability, and standardization of components (Straßburger, 2006).

The HLA specification was initially developed in 1995 by the U.S. Department of Defense Modeling and Simulation Coordination Office (DMSCO) together with Massachusetts Institute of Technology (MIT). The goal was to provide a standard for DOD simulation interoperability that would supplant the DIS protocol (McGregor, 2011).

The architecture, as it was planned, was for general purposes with no limitation in terms of specific simulation paradigms. The standard was based other general standards such as Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM). With this approach DMSCO ensured broad community support for their developed standard. Today, HLA is used not only in the U.S. military domain as a mandatory standard but also by most NATO members and other international DOD partners (Straßburger, 2006).

#### *a. Characteristics*

HLA standardizes the Application Programming Interface (API) rather than the communication protocol. The advantage of this approach is that the developers of simulation software can simply change HLA implementations as needed, as long as the changes remain consistent with the API specification (Figure 9).

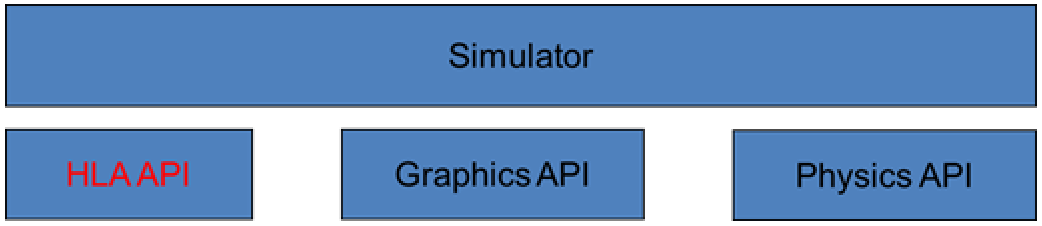


Figure 9. HLA API Structure.
Source: McGregor (2011b).

If, for example, a vendor comes up with a new scheme or a new way to reduce the usage of bandwidth, the vendor can implement that change easily. It does not influence the other parts of the software (McGregor, 2011b).

Another advantage of using HLA over DIS is that HLA provides a time synchronization mechanism to allow simulation with different time management schemes to support interoperability. In DIS every simulation is expected to run in real time. For example, simulating a tank moving from point A to point B will take the same amount of time as it would in the real world. This is good for special types of simulations, such as simulation of direct fire engagements. Nevertheless, if one needs to simulate a task like maintenance of a tank or an airplane, which can take weeks, a DIS-based simulation might be not the best option. HLA provides a time management functionality that allows it to coordinate time advancement in different simulations, which ensures that the flow of time remains consistent between the simulations that are working together. This allows for more complex scenarios to be simulated in a reasonable amount of time (McGregor, 2011b).

#### *b. Object Model Template*

HLA uses Object Model Templates (OMT) to describe and define software objects. These objects contain the information for all the different participants in the simulation environment. An OMT is independent from any vendor and any specific programming language. It has two types of structure tables (ST): the interaction-class ST and the object-class ST. Finally, an attribute table and a parameter table are part of an OMT. HLA OMTs are written and defined with Extensible Markup Language (XML). This ensures platform independence (You et al., 2016).

#### c. *The Run Time Infrastructure*

HLA interactions are mediated through a software component called the Run Time Infrastructure (RTI). All interactions with the simulation are controlled through the RTI. This architecture allows for more complex types of interaction and allows for more heterogeneous simulations to be connected because the RTI can translate the standards to be understood by different simulations. While the RTI provides an important capability, it comes with the cost that a complex software component needs to be included in all HLA simulation events.

### 2. Distributed Interactive Simulation

The Distributed Interactive Simulation (DIS) is a data protocol developed by a government and industry initiative in 1995 to describe how to connect various types of simulation systems at different locations all over the world. The main goal was to achieve complex and realistic virtual worlds for military simulation purposes in real time. The DIS protocol, which is designed to support a broad range of simulations and systems, allows it to be used in exercises that can include a combination of live, virtual, and constructive simulations.

"DIS draws heavily on experience derived from the simulation networking (SIMNET) program developed by the Advanced Research Projects Agency (ARPA), adopting many of SIMNET's basic concepts and heeding lessons learned" (IEEE, 2015, p. 8).

The advantage of DIS is that it is available without any licensing costs and restrictions. It is widely accepted, and everyone has permission to use and implement it. Consequently, each piece of simulation software that adheres to the IEEE standard can exchange data with other implementations. This is mainly achieved by DIS defining different packet types — so called "Protocol Data Units" or "PDUs." Each of those PDUs contains different data depending on its purpose: Entity State PDU, Collision PDU, Fire PDU Detonation PDU, and many more (McGregor, 2011a). A structural overview of the different PDU types is shown in Figure 10.

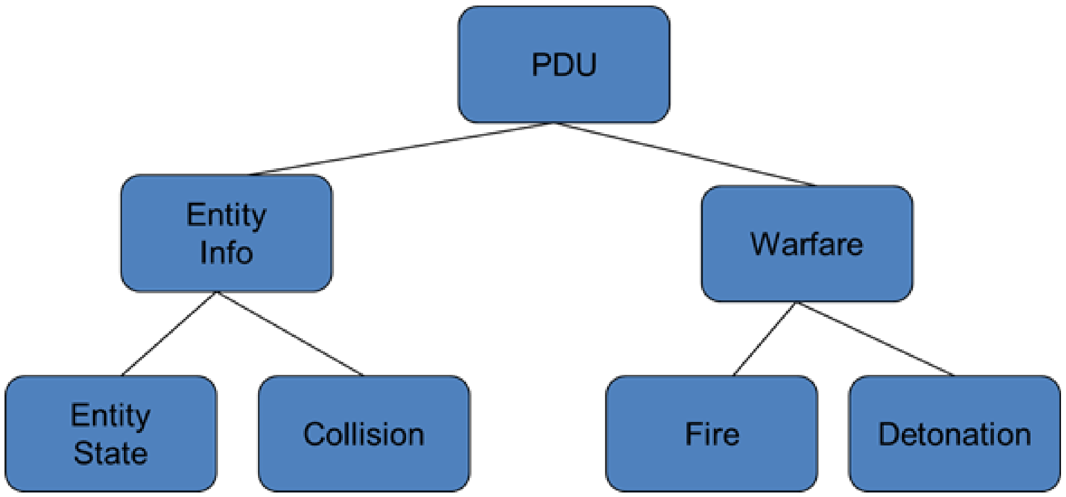


Figure 10. PDU Hierarchy. Source: McGregor (2011a).

#### *a. Different Types of PDUs*

In this section the types of commonly used PDUs and their respective purposes are described. Details about all of the theoretically different types of PDUs can be found in the IEEE Standard for Distributed Interactive Simulation—Application Protocols (IEEE, 2012, p. 51ff).

##### (1) Entity State PDU

An Entity State PDU (ESPDU) is the most common PDU. It is used whenever the other hosts must be informed about the actual state of a single entity at a specific time. Sending a ESPDU could be triggered by the change of the position, orientation, status of the entity, speed, or something similar.

##### (2) Fire PDU

A Fire PDU (FPDU) is used to inform the other simulations that an entity has fired on a target. Therefore, the entity ID of the shooter and the entity ID of the target are part of the content. Also, the launch location, fire rate, type of ammunition used, velocity, range, and a few more data points are included in this type of PDU.

(3) Detonation PDU

A Detonation PDU (DPDU) is used to inform that the detonation of a munition occurred. This can happen if an entity was hit by a direct fire munition or if some other ordnance explodes, such as an artillery round hitting the ground. Usually, this is the result of a Fire PDU. This type of PDU contains information such as the ammunition used, shooter ID, location, and detonation result.

(4) Collision PDU

A Collision PDU (CPDU) is issued by an entity when a collision occurs between the entity itself and another object, which is part of the simulation. Usually, both entities involved should launch a CPDU for the same event. This packet contains information about the IDs of the involved entities, the weight and velocity vector of the issuing entity, and—of course—the location of the event.

### *b. Basic Structure*

Each DIS-based PDU starts with a 96-bit-long header. Table 4 displays the different content contained within a PDU.

Table 4. PDU Header. Source: McGregor (2011a)

| Field size (bits) | Field name | Data type |
|---|---|---|
| 8 | Protocol Version | 8-bit enumeration |
| 8 | Exercise Identifier | 8-bit unsigned integer |
| 8 | PDU Type | 8-bit enumeration |
| 8 | Protocol Family | 8-bit enumeration |
| 32 | Timestamp | 32-bit unsigned integer |
| 16 | Length | 16-bit unsigned integer |
| 8 | PDU Status | 8-bit record of enumerations |
| 8 | Padding | 8-bit unused |

The field “Protocol Version” describes the release of DIS used. “Exercise Identifier” is a unique ID for each running simulation. This enables running multiple simulations at the same time and on the same network without interference. “PDU Type” explains the specific type of PDU content following the header (Figure 10). Some examples include: PDU type 1 stands for an EntityPDU, 2 for a FirePDU, 3 for a DetonationPDU, 4 for a CollisionPDU, and so on. A complete list of different PDU types can be found in the IEEE Std 1278.1-2012 paper, section 5.2.4. Theoretically, there are 192 different types of PDU possible (0–191); however, only 72 of them are used. The numbers from 192 up to 255 are reserved for testing purposes. “Protocol Family” indicates the kind of group to which this PDU belongs (Figure 10, second row). The field “Time Stamp” is the time the PDU was sent expressed in milliseconds. It could be used for duplicate packets and out-of-order packet as well. “Length” simply describes the length of the PDU in bytes. The status of a PDU is identified in the field “PDU Status,” and if padding was used, this will be described in the last field, “Padding” (IEEE, 2012).

#### c. *Network Architecture*

A typical DIS infrastructure is a peer-to-peer (P2P) network with no central server unit(s). Every participant has equal rights and sends and receives PDUs. According to the entity ID, the client decides if this packet is important for its simulation or not. If there is more than one simulation running on a specific network, subnetworks are usually used. However, it is also possible to run different simulations within the same address space. The exercise identifiers help to decide which simulation a PDU belongs to. A heartbeat of 5–10 seconds is used for mobile entities to send their status updates. Static entities, such as minefields, may require a longer time period (~60 seconds).

The use of a P2P structure instead of a client-server-based architecture makes it necessary for participants to trust each other. DIS is cooperative in that it assumes no one is trying to spoof the system, issue fraudulent PDUs, ignore damage results, and so on (McGregor, 2011a, p. 51).

## D. ASSESSMENT AND IMPLICATIONS FOR THE DEVELOPMENT OF THE PROTOTYPE

Based on the investigation results presented so far in this chapter and the given assumptions of this research project, we can assess the results in the following paragraphs and devise a solution for the development of the prototypes. This includes the assessment of existing software products as well as the type of network communication structure and the protocol to use.

### 1. Existing Software Products

Two major software products were investigated in Chapter II.A. ATAK is a very popular and powerful application. Because some of the versions are not available for the German military and the whole package is too big for the needed use cases, however, the decision was made to create our own lightweight and tailored-to-the-problem software. Also, the second use case of showing real effects of virtual events is not doable with ATAK.

Similarly, KILSWITCH has potentially the same possibilities as ATAK—and therefore the same disadvantages. In addition, another serious disadvantage has arisen that almost certainly excludes the use of KILSWITCH. The security gaps discovered in the software, especially in live training, are unacceptable when used in sensitive military areas and lead to an enormous loss of confidence in the product.

Taking all these facts into account, it is clear there is currently no way around developing our own lightweight and secure software product. Such a solution is developed during the thesis and the general feasibility of the required functions is proven.

### 2. Network Communication Structure

The question of the correct connection to the existing system is not easy to answer. In Chapter II.B, the specifications of NFC, Bluetooth, Wi-Fi, USB, and 5G were examined and the advantages and disadvantages of each were identified. They are summarized in Table 5. Each of the examined technologies offers good reasons to use this method.

Ultimately, the choice fell on a connection via Wi-Fi. The range and bandwidth speak for Wi-Fi, which requires relatively low power consumption. Furthermore, Wi-Fi is well established, and it is easily handled in all modern programming languages and is widely used. Libraries are available and a lot of examples for this choice of technology can be found.

In the future, a connection via 5G must also be considered if the infrastructure has been expanded (especially on training facilities) and enough client devices are equipped with this technology.

Table 5. Comparison of Different Communication Technics

| | NFC | Bluetooth | Wi-Fi | USB | 5G |
|---|---|---|---|---|---|
| **Frequency** | 13.56 MHz | 2.4 GHz | 2.4 / 5 GHz | | 28 GHz |
| **Bandwidth** | 424 kbps | 480 Mbps | 9.6 Gbps | 40 Gbps | 10 Gbps |
| **Range** | 10 cm | 100 m | 250 m | 10 m | 250 m |
| **Power** | 10 mW | 100 mW | 1000 mW | 7.5 W | 1000 mW |
| | | | | | |
| **Pro** | power | | bandwidth | bandwidth | bandwidth |
| | frequency | | range | stable | range |
| **Con** | range | range | | range | frequency |
| | bandwidth | | | wired | infrastructure |

Since no real operational systems are available, a test setup comparable to the real system must be created. The communication system setup is represented by a configurable router of the Fritz! Box 7390 type from the German company AVM. The end devices can be connected via Wi-Fi. VPN connections can also be configured that may be needed in the further course of the process. Figure 11 shows the desired schematic structure as it would be used in live operation.

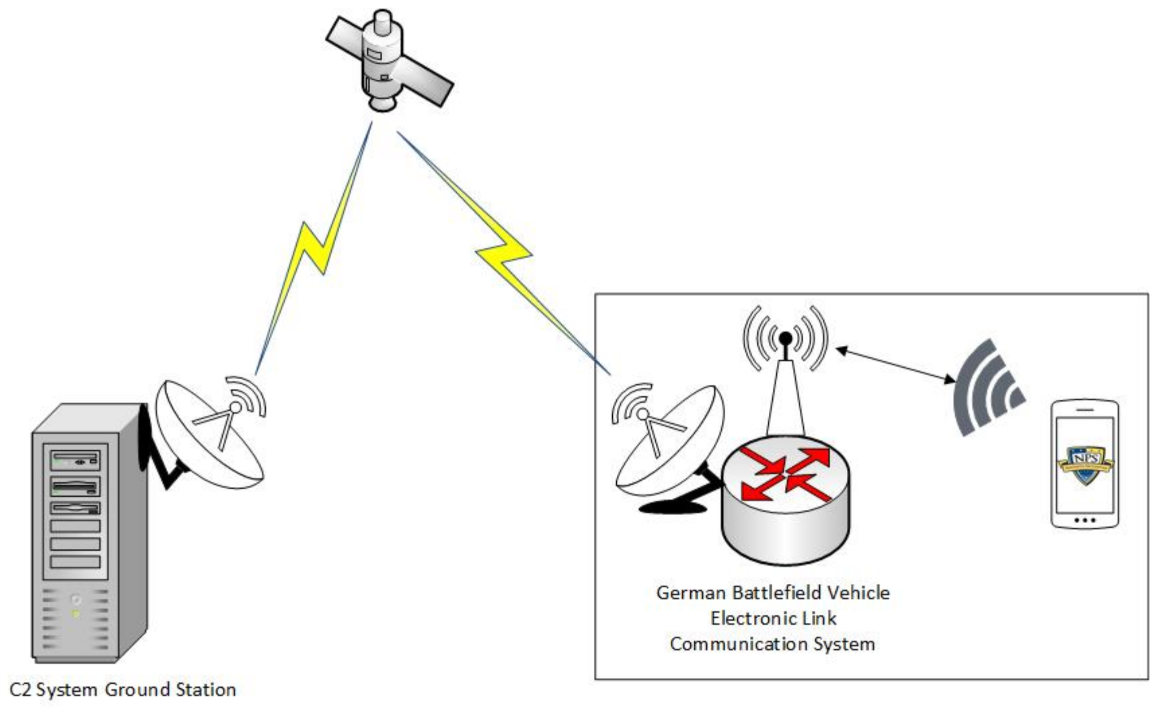


Figure 11. Network Schema for Actual Use

Figure 12 shows the structure of the test configuration. The C2 system is replaced by VR Forces version 4.8, the satellite connection by a simple Ethernet cable. The location of the German Battlefield Vehicle Electronic Link Communication System is mapped by the aforementioned router from AVM.

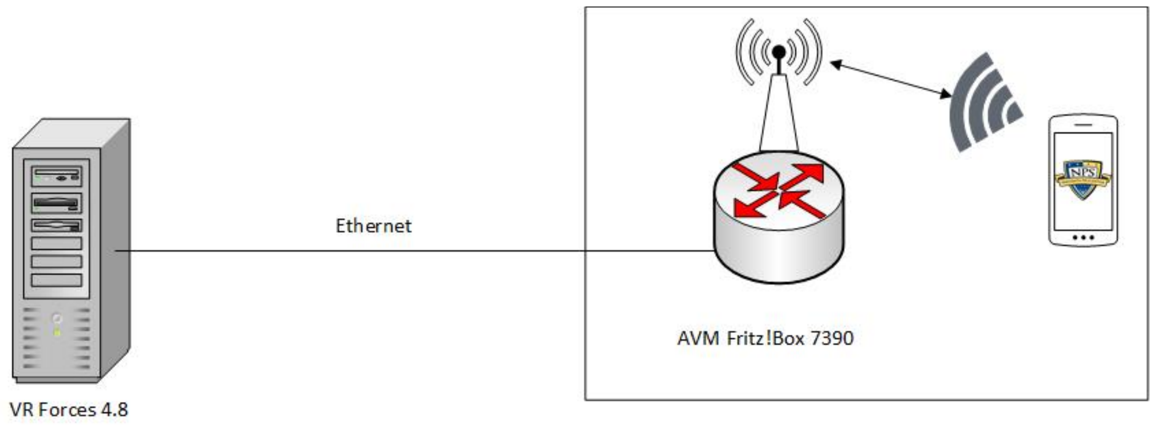


Figure 12. Network Schema for Test Configuration

The latency of a satellite connection, which can be more than one second, is neglected in the experimental setup. The aim is to transfer data from the mobile device via the network to the VR-Forces software in a time span of less than 0.1 seconds.

### 3. Protocol

The question of whether HLA or DIS should be used can be answered very easily. The respective advantages and disadvantages were presented in detail in Chapter IIC. Further, the assumption made in section E.2 of Chapter I that the German C2 system uses the DIS protocol defines which architecture the application must use.

### 4. Summary

In summary, in order to achieve the desired goal of improving live training, a separate software solution for mobile devices must be developed. The devices should be connected with wireless technology using Wi-Fi. The developed software must be able to send all simulation-relevant data to a C2 system using the DIS protocol. This C2 system is simulated in the test setup by the VR-Forces software version 4.8.

THIS PAGE INTENTIONALLY LEFT BLANK

# III. AVAILABLE PROGRAMMING LANGUAGES

The Android system is an operating system (OS) as well as a software platform based on a Linux kernel. Several programming languages are available for this platform, and they can be used to develop applications to run on mobile devices that use an Android OS. Based on the recommendation of Google, the Android Studio Software Development Kit is used for the work in this thesis. This SDK primarily supports two different programming languages—JAVA and KOTLIN. In this section we analyze the pros and cons of both. Based on this analysis, a decision is made about which programming language should be used to create the application.

## A. JAVA

Java is an object-oriented programming language. The first version was developed by James Gosling at Sun Microsystems in 1995. Oracle acquired Sun, and with it Java, in 2010. Since its original development there have been several development strands. Currently the version from Oracle is seen as the "official" version, but there are several other open-source versions. The best known of these is probably OpenJDK (Java Development Kit). There are also Java versions that have been optimized for specific hardware. One example is the LiberiaJDK, which has been specially optimized for operation on the ARMv7 architecture of a Raspberry Pi.

The original purpose of this new programming language was to implement the so-called WORA (write once, run anywhere) principle. For this purpose, the source code written in Java is not translated directly into machine-readable machine language by a compiler such as C or C ++, but an intermediate step is taken. The Java Compiler translates the source code into Java bytecode. This is then interpreted and executed at runtime by a Java Runtime Environment (JRE) with a Java Virtual Machine (JVM) on the target system (Figure 13). Although this has a small runtime performance cost compared to other compiler languages, it has the great advantage that this Java bytecode can be executed on almost any hardware with a running JVM—regardless of which operating system or processor architecture is located there (Arnold et al., 2005).

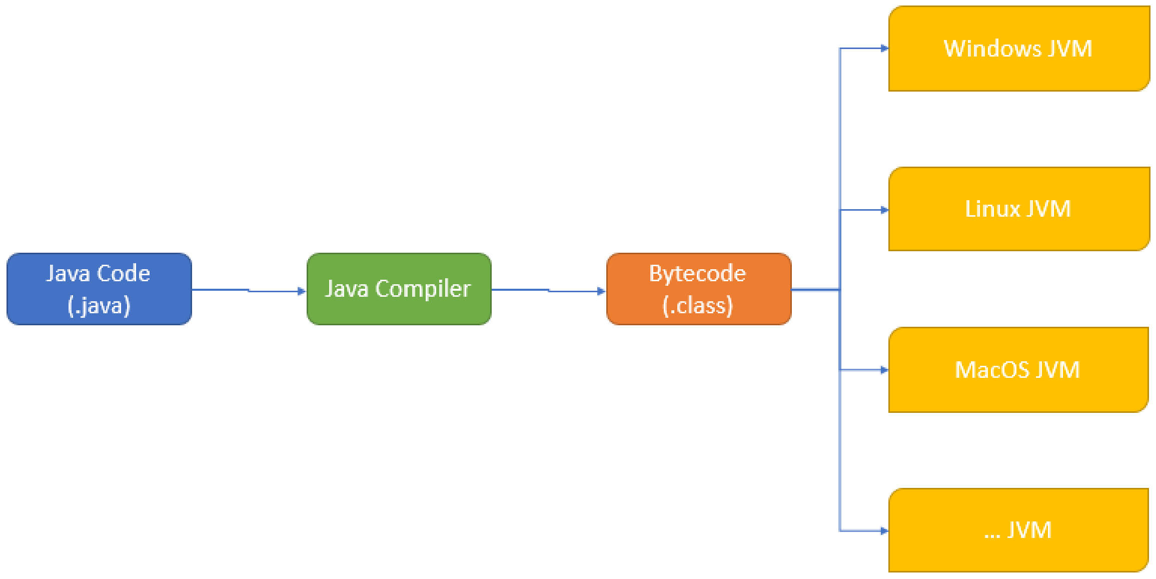


Figure 13. Java Compiling and Running Process

Today, Java is one of the most popular programming languages in the world. The TIOBE index lists Java as one of the two most frequently used programming languages worldwide since 2001 (*TIOBE*, n.d.).

## B. KOTLIN

KOTLIN is a platform-independent programming language. It was introduced in 2011 as a new language for the JVM. It was developed primarily by the company JetBrains, which has its headquarters in Saint Petersburg, Russia. The name "Kotlin" comes from an island that lies offshore in the Baltic Sea. The first stable release (1.0) was published in 2016 and has grown rapidly since then.

Kotlin offers the possibility to integrate both Java and JavaScript source code and to use existing software libraries of these two languages. The Kotlin compiler translates Kotlin source code is translated into bytecode, which can then be executed by a JVM on various operating systems (Figure 14).

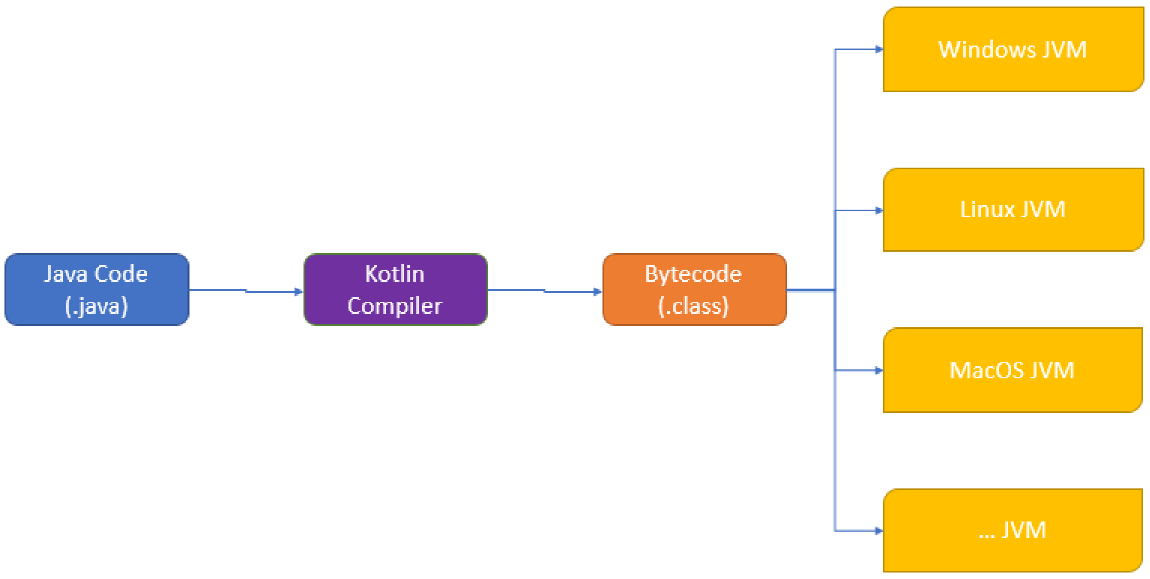


Figure 14. Kotlin Compiling and Running Process

The advantages of Kotlin over Java are that Kotlin is faster and less error-prone programming. In Kotlin variables do not have to be encapsulated by the programmer as they do in Java (private attributes, public getter and setter). This encapsulation is done in the background by the development environment and the programmer does not have to worry about it. When software is prone to errors at runtime, Kotlin has the particular advantage that, for example, no null pointer exceptions can occur. Such errors often lead to unpredictable crashes in Java code (Bose, 2018).

Google has supported Kotlin in version 3.0 since 2017. At the I / O 2019 developer conference, Google declared Kotlin the preferred programming language for developing applications after a copyright litigation with Oracle.

## C. COMPARISON, EVALUATION, AND RATING

Both the Java and Kotlin programming languages are suitable for developing applications for mobile devices based on Android OS. Yet, Kotlin offers the possibility of using all the advantages of Java while avoiding some of the disadvantages (extensive source code, susceptibility to errors due to null-pointer exception at runtime).

It is also possible that in the future Google will completely prohibit the use of Java for developing applications for Android devices. Looking ahead and considering the advantages of Kotlin just mentioned, it appears that the best option is to use Kotlin to develop new apps. Therefore, it is used in this thesis as the primary programming language for code, provided that it is to be executed on Android-based devices.

# IV. IMPLEMENTATION AND DEMONSTRATION

## A. MOBILE ENTITY SIMULATOR

As was concluded in the previous chapter, new software must be developed to fulfill the need for a more realistic and secure simulation solution for training in the German military. The proposed software should run on an Android-based mobile device equipped with a GPS antenna and a Wi-Fi interface. The programming language for this solution is Kotlin.

The software should be able to send ESPDUs with their own position data and selected signature (military symbol) over an IP-based network to a running VR-Forces machine, which simulates the German C2 system. For the basic setup three devices are used: a notebook for development, a smartphone as the mobile device on which the software is running, and a powerful computer on which VR Forces is running. In the first step, these three devices are connected via Wi-Fi, and they also have a connection to internet (Figure 44).

### 1. Equipment Used and Setup

A standard development environment was used in the development of the prototype system. This setup included a Lenovo notebook computer as the development environment, a Samsung Galaxy J7 for the mobile device, and a Ryzen 7 to run VR Forces. The development was done using Android Studio as the IDE. The details of the configuration used can be found in Appendix A.

### 2. Prototype Development

To make an application usable, a well-designed human interface is important. For a prototype, a simple layout is sufficient, but it still should be easy and intuitive to use. The screen before this prototype is divided into three main parts (Figure 15). Part one at the top-center displays different military symbols from which to choose. For a demonstration, four to six different symbols (at least for friendly and enemy units) are enough. In the production version all possible military vehicle symbols must be available. Part two is for input and output of IP addresses and debug information. This part will not be important in the production version since it is not expected that the end user will need to use this functionality. Part three contains controls to start and stop the transmission of data from the device.

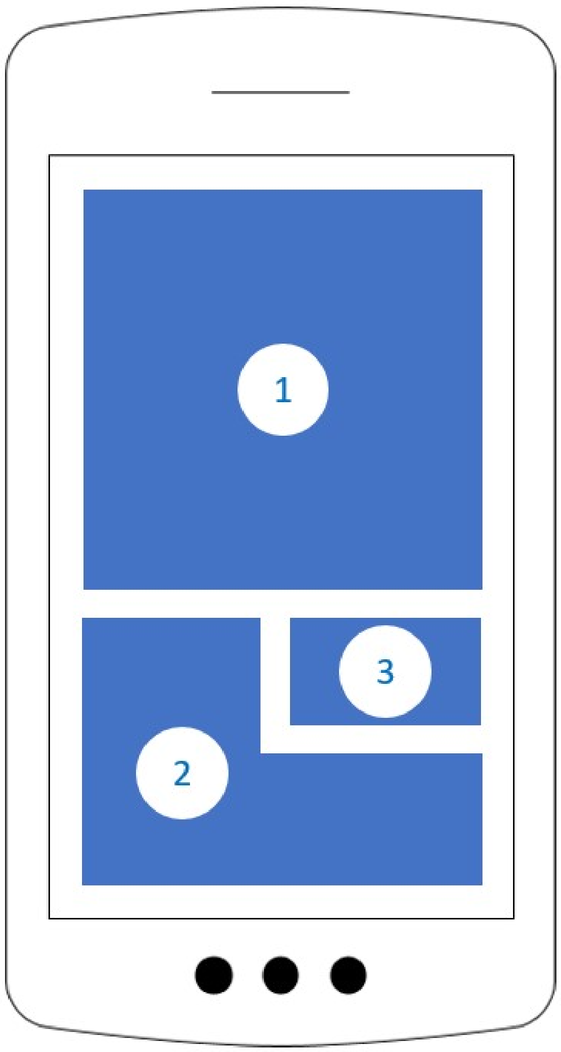


Figure 15. Design of Human Interface for Mobile Entity Simulator

There are several problems to be addressed in order to develop a prototype that meets the requirements. The division of the development process into subcomponents will not only facilitate that process but also further testing. The following components and their implementation are described in this chapter. Figure 16 shows in a flow chart how the application will be used. After starting the app, the user selects a military symbol which he/she would like the system to display in the C2 systems. Then an IP address and port number can be inserted. If the process of sending one's position needs to be started, a co-routine starts working. First, a socket connection is established. Second, the location of the mobile device is determined. Third, an entity state PDU is created, and fourth, the PDU is repeatedly sent at a given time interval. After the user decides to stop sending PDUs, the application goes back to the start screen and the user can change the selected military symbol and/or the IP and port number, and/or simply restart sending packages.

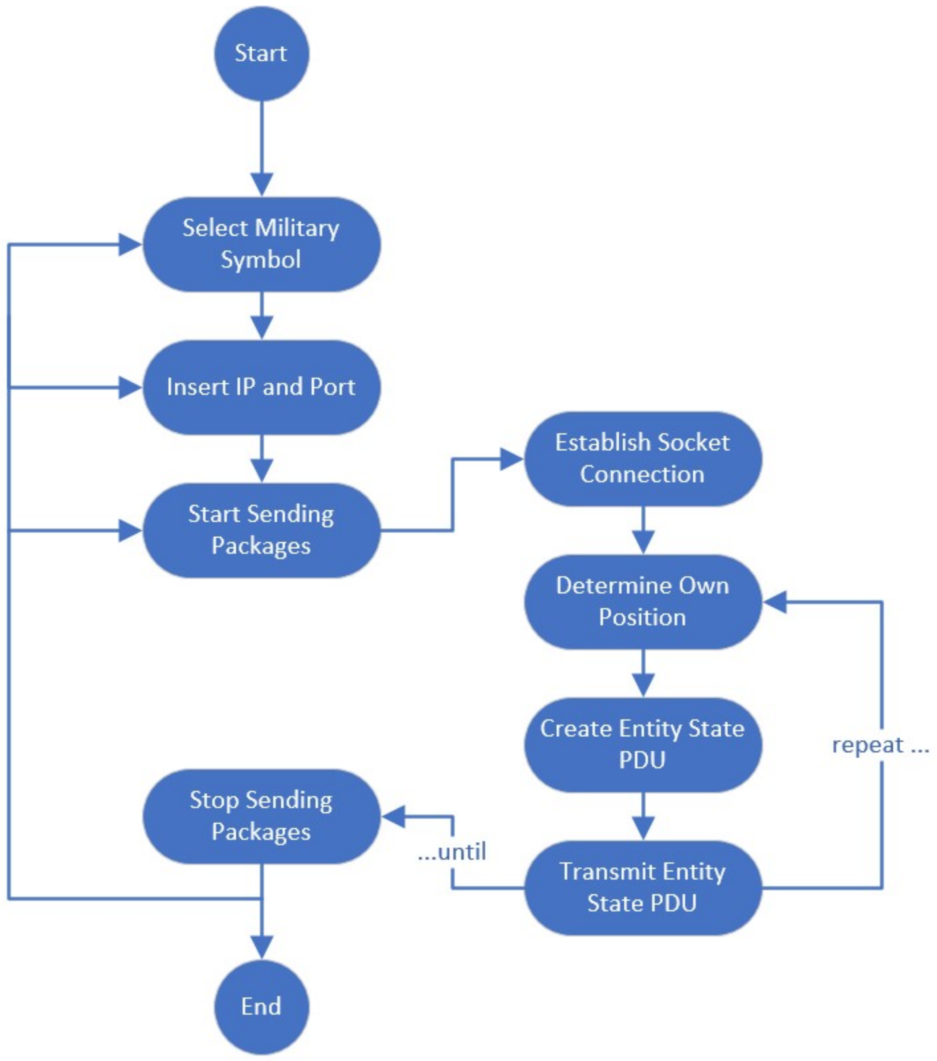


Figure 16. Flow Chart of Mobile Entity Simulator

### a. *Select a Military Symbol*

After starting the application, the user has the option of choosing from a range of different military symbols. The selection is made by clicking the appropriate button (Figure 17). As feedback, the button changes color to a darker shade. The implementation technique is explained in the next paragraph.

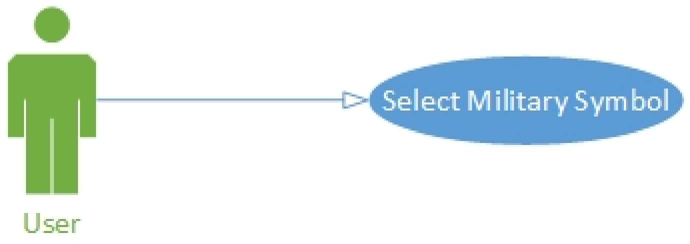


Figure 17. Mobile Entity Simulator Use Case—Select Military Symbol

First, images for those buttons must be produced. Military symbols are available from several sources. For this work the original U.S. Department of Defense (DOD) document was used (Department of Defense, 2008). To create the icon for the "button pressed" status the original symbol was processed by a graphic program (Gimp). Both images are inserted into the project in the "res/drawable" folder.

To display a symbol on the application surface it needs to be inserted in a layout XML file ("res/layout" folder). The following snippet of code shows the layout descriptive for a blue force tank:

```
<ToggleButton
    android:id="@+id/button_NotPressed_SPz_blue"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_row="1"
    android:layout_column="0"
    android:background="@drawable/spzblue_button"
    android:textOff=""
    android:textOn="" />
```

A ToggleButton was chosen because it allows for different status types (pressed, not pressed). The first line says what name the image icon has. The next two lines describe the size of the image within the whole layout, and the following lines specify at which position the image will be placed. The following attribute tells the system where to find the source image, and in the last two lines, alternative texts can be inserted.

This element can then easily be referenced in the associated Kotlin class.

```
val button_spzblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_SPz_blue)
```

This is done for five more symbols in this prototype demonstrator and the outcome is shown in Figure 18.

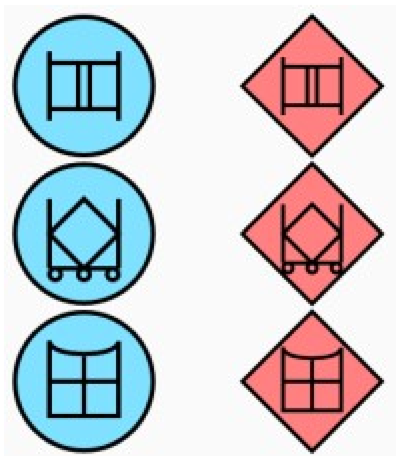


Figure 18. Layout of Military Symbols

Each of these buttons is observed by an action listener. In this specific case an OnSetCheckedChange listener was chosen (see code below). When an action event is recognized by the listener, it checks to see whether the button was selected. If so, all the other buttons are unchecked. This ensures that only one button per time can be selected (alternatively, a button group or spinner could be used here).

```
button_spzblue.setOnCheckedChangeListener { _, _ ->
    if(button_spzblue.isChecked) {
        button_pzmediumred.isChecked = false
        button_sanblue.isChecked = false
        button_sanred.isChecked = false
        button_pzmediumblue.isChecked = false
        button_spzred.isChecked = false
    }
}
```

At this point the actor (user) can choose from six different types of military symbols—three blue forces and the corresponding three red forces.

#### *b. Enter IP Address and Port Number*

After choosing the military symbol by which the application should be recognized in the C2 system, it is necessary to specify an internet protocol (IP) address and a port number (Figure 19). Depending on the structure of the training and simulation environment, a single IP could be chosen as well as a multicast address. The port number is the port on which the C2 system expects the packages and on which to listen.

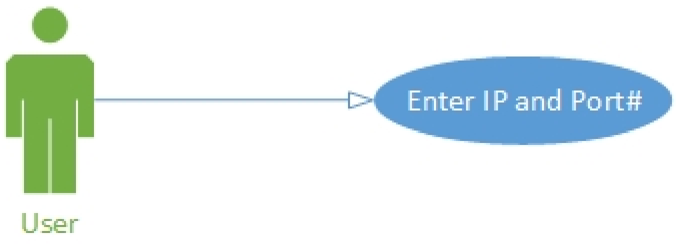


Figure 19. Mobile Entity Simulator Use Case—Enter IP and Port Number

For optimal input and human interface design, simple text boxes that the user can easily edit were chosen. The text fields in this example are preconditioned for the testing setup. The broadcast address used is 192.168.188.255 and the port number is 3000. This code is located at the same spot as in the previous use case.

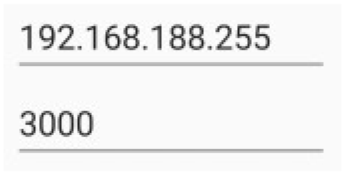
```
<EditText
    android:id="@+id/ipTextField"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_row="9"
    android:layout_column="0"
    android:ems="8"
    android:inputType="textPersonName"
    android:text="192.168.188.255" />

<EditText
    android:id="@+id/portTextField"
    android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:layout_row="10"
android:layout_column="0"
android:ems="8"
android:inputType="number"
android:text="3000" />
```

To get access in the Kotlin code, the two text fields must be referenced first. This handled in the same manner as the military symbol buttons just described (see the following snippet of code).

```
val ipAddress = view.findViewById<EditText>(R.id.ipTextField)
val port = view.findViewById<EditText>(R.id.portTextField)
```

The resulting layout is shown in Figure 20. At this point, the software can access the information about where to send the PDUs. This becomes important in the use case "Establish a Socket Connection" (d).


192.168.188.255

3000

Figure 20. Layout IP Address and Port Number

### c. *Start Sending Packages*

After a military symbol is selected and the IP address and port number are inserted, the next step according to the flow chart (Figure 16) is to start the sending process. For now, this is also the last use case in which the user is an actor (Figure 21). The next user-driven use case will be "Stop Sending Packages," which is described later.

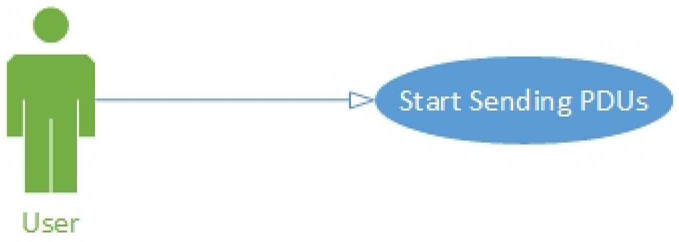


Figure 21. Mobile Entity Simulator Use Case—Start Sending PDUs

For this use case, a simple button was chosen to enable the user to give the application the signal to start sending PDUs. For the layout, the following code was inserted in the known XML file.

```
<Button
    android:id="@+id/startButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_row="3"
    android:layout_column="1"
    android:text="Start" />
```

This code creates a simple gray colored button with the text "Start" on it. By default, the button changes color when the user clicks on it. In the Kotlin code, it needs to be referenced similar to the previous examples.

```
val startButton = view.findViewById<Button>(R.id.startButton)
```

Like the buttons for choosing a military symbol, this start button needs an observer which reacts to user inputs.

```
startButton.setOnClickListener {
    if(stopButton.isActivated)
        stopButton.isActivated = false
    start()
}
```

When a click on the button is recognized, it simply deactivates the stop button and calls a function (here: start()). This establishes the execution of the following four use cases and ends when the user clicks the button "Stop" (as described in the use case "Stop Sending Packages").

### d. *Establish a Socket Connection*

This is the first use case involving a co-routine. It is also the first use case in which the actor is the application itself. The main task in this use case is to build an entity state PDU and send it to the given IP address and port (Figure 22). This action must be done repeatedly until the user presses "Stop" and ends the co-routine. This co-routine must run parallel to the main program; otherwise, the graphical user interface (GUI) will be blocked and the user will be unable to press the button. To force the software to run in parallel, we use the thread technique for the Java util library. Threads allow a program to run multiple parts at the same time. The "start()" call seen in the part before must be a runnable thread for this purpose.

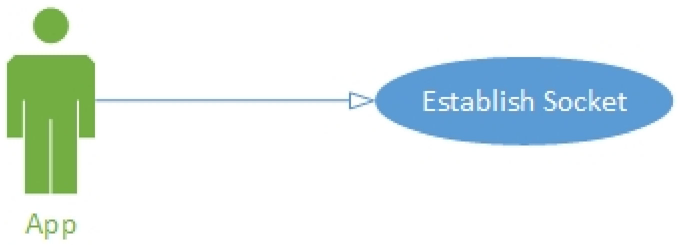


Figure 22. Mobile Entity Simulator Use Case—Establish Socket

The first use case in this co-routine is to establish a connection to the C2 system by opening a socket. A "DatagramSocket" was used and set into "broadcast" mode. Then the IP address and port number taken from the text fields become parameters for the constructor call to create a "DatagramPacket" object (see the following code).

```
val datagramSocket = DatagramSocket()
datagramSocket.broadcast = true

// get IP and Port from text field

var ina  = InetAddress.getByName(ipAddress.text.toString())

var port = Integer.parseInt(port.text.toString())

// create DatagramPacket
```

```
          var dp2  = DatagramPacket(…., …., ina,
          Integer.parseInt(port.text.toString()))
```

An established socket connection is necessary to continue with the next steps. Since this part is very error-prone, any problems that occur must be intercepted and, if necessary, processed. For this purpose, the entire code block is surrounded by try-catch brackets. Errors that occur in the try block can be handled in the catch block. In this prototype version, the error treatment is just to send the message to a debugger and the command output line.

### *e. Determine Own Position*

Once the socket is established, we need the actual position of the mobile device. For that purpose, it is necessary to provide access to the hardware (Figure 23).

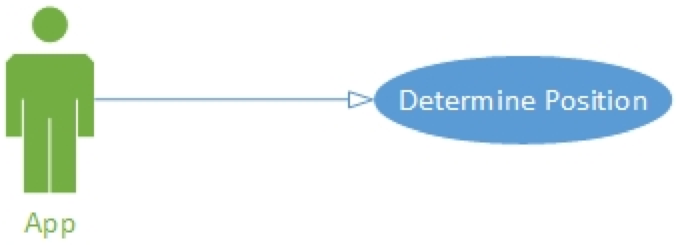


Figure 23. Mobile Entity Simulator Use Case—Determine Position

This use case must be done in different steps. First, the project must be enabled to use the hardware (in our case the GPS antenna of the mobile device). Therefore, a new dependency implementation must be inserted in the "build.gradle" file, which is located in the "Gradle Scripts" folder. The following line is used for this purpose:

```
dependencies {
          …
          implementation 'com.google.android.gms:play-services-location:17.1.0'
          …
}
```

Now the main program can read the actual GPS position, registering itself as a listener to the antenna. Whenever the location changes, the program receives a call-back and the variables are set with the actual coordinates in 3D (latitude, longitude, and altitude).

```
// to get access to the GPS data of the mobile device
private var fusedLocationProviderClient: FusedLocationProviderClient? =
null
override fun onStart() {
    registerForLocationUpdates()
}
override fun onStop() {
    unregisterForLocationUpdates()

}
fun updatePosition(location: Location) {
    latitude = location.latitude
    longitude = location.longitude
    altitude = location.altitude
}
@SuppressLint(“MissingPermission”)
fun registerForLocationUpdates() {
    val locationRequest = LocationRequest.create()
    val looper = Looper.myLooper()
    locationProviderClient.requestLocationUpdates(locationRequest,
locationCallback, looper)
}
private fun getFusedLocationProviderClient():
FusedLocationProviderClient {
    if (fusedLocationProviderClient == null) {
        fusedLocationProviderClient =
            LocationServices.getFusedLocationProviderClient(activity!!)
    }
    return fusedLocationProviderClient!!
}
private val locationCallback: LocationCallback = object :
LocationCallback() {
    override fun onLocationResult(locationResult: LocationResult) {
        super.onLocationResult(locationResult)
        val lastLocation: Location = locationResult.lastLocation
        updatePosition(lastLocation)
    }
}
```

Now the GPS coordinates data are available in the program, Unfortunately, GPS uses another reference system than DIS does. GPS uses the so-called “World Geodetic System 1984 (WGS84)” while DIS uses the “Earth-Centered Cartesian Coordinate System (ECCCS).” Therefore, the received values must be converted. Thankfully, the openDIS library used for this project offers a class called “CoordinateConversions” that translates WGS84 into ECCCS coordinates (see the following code).

```
var disCoordinates = CoordinateConversions.getXYZfromLatLonDegrees(
    latitude,
    longitude,
    2.0
)
```

The method getXYZfromLatLonDegrees() returns a 3DVector object containing the coordinate in ECCCS, which is used by all DIS-based simulation and C2 systems. Since we only deal with land-based vehicles in this program, a centered height of 2.0 m was assumed.

#### *f. Create the Entity State PDU*

DIS requires and enables the transmission of a lot of information about an entity within one package (Figure 24). Position is only one piece of that information. Depending on the specific scenario, additional data are required. For this thesis some more are needed.

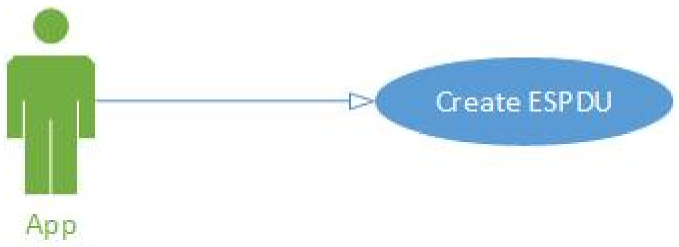


Figure 24. Mobile Entity Simulator Use Case—Create ESPDU

DIS can work in peer-to-peer as well as in client-server configurations. Consequently, different simulation systems can be connected and share information. Therefore, three basic attributes must be included in the packets: a side ID (for multiple-location simulations), an application ID (which specifies the application used), and an entity ID (which is the unique identifier within an application). For the purpose of the prototype developed in this thesis, these attributes are simply set to “1” (see the following code).

```
entityID.siteID = 1.toShort()
entityID.applicationID = 1.toShort()
entityID.entityID = 1.toShort()
```

Depending on which military symbol was chosen in use case “Select a Military Symbol,” the specific type of entity must be defined. The information about the nation to which this vehicle belongs; what type (platform) it is, whether it is land, sea or aircraft; to which category the entity can be assigned; and other sub-categories is specified. In addition, the entity can also be assigned a readable name, which can be displayed in some simulation systems (such as VR Forces) and C2 systems. In the following section of code section, the setting of the data for a medic vehicle for a U.S. platform and a Russian platform, respectively. is shown as an example.

```
} else if (button_sanblue.isChecked) {
    forceID = ForceID.FRIENDLY
    entityType.country = Country.UNITED_STATES_OF_AMERICA_USA
    entityType.entityKind = EntityKind.PLATFORM
    entityType.domain = Domain.inst(PlatformDomain.LAND)
    entityType.category = 2.toByte()
    entityType.subCategory = 38.toByte()
    entityType.specific = 1.toByte()
    marking.characters = “SanBoxer”.toByteArray()
} else if (button_sanred.isChecked) {
    forceID = ForceID.OPPOSING
    entityType.country = Country.RUSSIA_RUS
    entityType.entityKind = EntityKind.PLATFORM
    entityType.domain = Domain.inst(PlatformDomain.LAND)
    entityType.category = 2.toByte()
    entityType.subCategory = 7.toByte()
    entityType.specific = 21.toByte()
    marking.characters = “MT-LB ambulance”.toByteArray()
```

In the previous use case, the entity’s position was determined and converted to DIS understandable format. This data can be easily added to an entity state PDU as shown in the following snippet of code.

```
espduLocation.x = disCoordinates [0]
espduLocation.y = disCoordinates [1]
espduLocation.z = disCoordinates [2]
espdu.entityLocation = espduLocation
```

The last step is to add a time stamp to the package. This is important because the network structure setup used and the disparately located simulation or C2 systems cannot guarantee that DIS packages will be received in the correct order. However, since every PDU has a time stamp, packets arriving later or subsequently can be treated accordingly. For example, in live training, the vehicle's position at time t-1 is no longer relevant if the position is already displayed at time t. The class "DisTime" from openDIS library makes it easy to get the right format.

```
var timeStamp = DisTime().disAbsoluteTimestamp
espdu.timestamp = timeStamp
```

At this point the PDU is packed and ready to send.

#### *g. Transmit Entity State PDU*

The last use case in the co-routine is sending the build PDU package to the given IP address and port number (Figure 25).

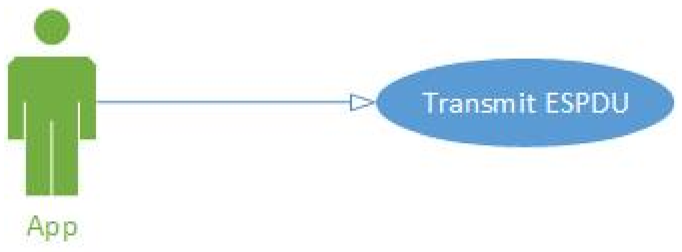


Figure 25. Mobile Entity Simulator Use Case—Transmit ESPDU

The produced entity state object is marshaled to a byte array and sent through the opened socket to its receiver(s).

```
val datagramSocket = DatagramSocket()
datagramSocket.broadcast = true

val baos = ByteArrayOutputStream()
val dos = DataOutputStream(baos)

espdu.marshal(dos)
val data = baos.toByteArray()
var dp2 = DatagramPacket(data, data.size, ina, port)
datagramSocket.send(dp2)
```

At this point one entity state PDU has been produced and sent via a socket connection. The whole code for this co-routine is summarized in the function "sendOnePacket(…)." Because we need to update the status of a vehicle in a regular manner, this function is called in a loop repeatedly with a pause of one second.

```
while (!done) {
    sendOnePacket(view)
    Thread.sleep(1000)
}
```

This architecture, which employs thread technology, enables the user to interact with the application while the loop is running. The user can change the type of vehicle simply by selecting another military symbol. This new information is built in the next package. Therefore, the simulation system can change the attribute of the shown entity within one second if there is no delay in the network.

### *h. Stop Sending Packages*

The described co-routine use cases in the preceding sections will run unlimited times if there is no action taken by the user.

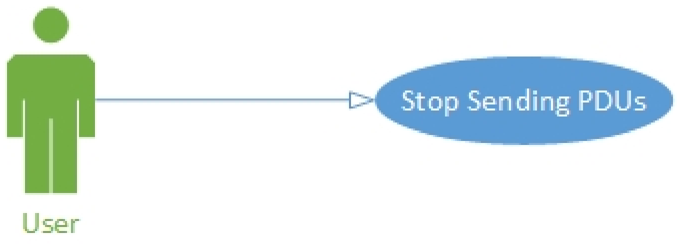


Figure 26. Mobile Entity Simulator Use Case—Stop Sending PDUs

Of course, this has to be enabled, and for this purpose a new button—similar to the start button—is inserted.

```
<Button
    android:id="@+id/stopButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_row="4"
```

```
android:layout_column="1"
android:text="Stop" />
```

This code creates a simple gray colored button with the text “Stop” on it. By default the button changes the color when the user clicks on it. In the Kotlin code, this button needs to be referenced as in the previous examples.

```
val stopButton = view.findViewById<Button>(R.id.stopButton)
```

This button is also observed by a listener. If a user presses “Stop,” a function called “shutdown()” within the running thread is called. The function itself simply sets a boolean variable called “done” to false. Doing this, the condition for the while-loop in use case “Transmit Entity State PDU” is not given anymore and the thread stops running.

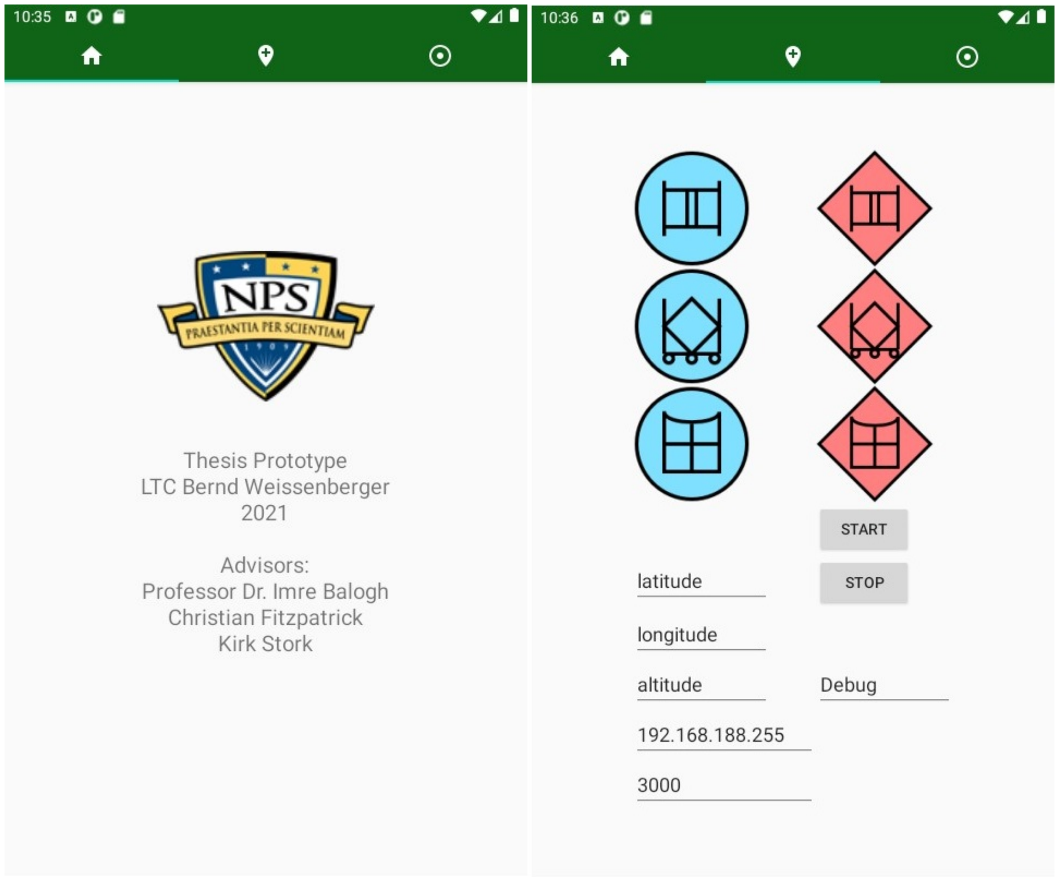
According to the flow chart shown in Figure 16, the user has three options: change the military symbol used, change the IP address and/or port number, or simply restart sending PDUs with the same settings.

### 3. Summary of the Development Process

Using a new programming language to develop a new application from scratch is challenging. The setup of both the hardware and the software has to be perfect for the task. Different demands from different libraries on the Java version to be used, for example, can lead to lengthy configuration processes. The connection of a mobile device to the development machine also initially led to difficulties (as described in section A.2 of Chapter V) but was necessary because testing Android applications only with the built-in emulator might not reveal all errors.

The resulting product is now able to send DIS-compliant data packets via a network to an adjustable address (IP and port number). The type of vehicle can be selected by simply clicking on the corresponding military symbol, and the current position of the mobile device is automatically inserted into the data package. The transfer of this information to the C2 or the simulation system takes place continuously every second.

Figure 27 shows the resulting user interface. On the left, the start screen is shown, and on the right, the main fragment of the software. Because this is a prototype version, debugging information is available (according to the design shown in Figure 15). The important source code files are included in Appendix B of this work.



Thesis Prototype
LTC Bernd Weissenberger
2021

Advisors:
Professor Dr. Imre Balogh
Christian Fitzpatrick
Kirk Stork

latitude
longitude
altitude
192.168.188.255
3000

Figure 27. Screenshot of Entity State Simulator

### 4. Testing and Demonstration

Of course, micro and mini tests were constantly carried out during the development process to ensure that every section of code and every use case worked as intended. This section describes the tests of the software as well as the integration into a test scenario. The first step is to check whether each device in the setup has an IP connection to every other device involved. This is done by sending "ping" commands between the devices. If there is a connection, the target device answers with a "response." In Figure 28, the "ping" from the VR Forces computer to the mobile device is shown as an example.

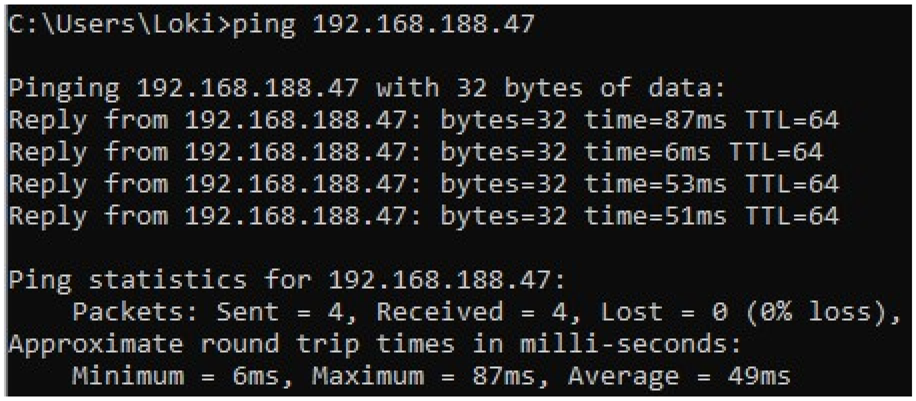

```
C:\Users\Loki>ping 192.168.188.47

Pinging 192.168.188.47 with 32 bytes of data:
Reply from 192.168.188.47: bytes=32 time=87ms TTL=64
Reply from 192.168.188.47: bytes=32 time=6ms TTL=64
Reply from 192.168.188.47: bytes=32 time=53ms TTL=64
Reply from 192.168.188.47: bytes=32 time=51ms TTL=64

Ping statistics for 192.168.188.47:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 6ms, Maximum = 87ms, Average = 49ms
```

Figure 28. Connection Test—Ping

This response shows that there is a stable connection. This can be recognized on the one hand by the fact that none of the four packets has been lost because four packets are received again. What becomes clear also is that the runtime within the network is acceptable (between 6 ms and 87 ms). After the connection is confirmed, the route of the packets is checked. If the setup works correctly, each packet should run via the AVM router, since the network is in infrastructure mode (as described at II.B.3.b(2)).

This status can be checked by using a command called "tracert." It does something similar to the "ping" command. It sends packages to a destination and receives a response. Additionally, tracert tracks the path of packages. In Figure 29 the execution of the command on the VR Forces computer is shown trying to reach the mobile device (IP: 192.168.188.47). In step 1 the response of the AVM router is shown (IP: 192.168.188.1), and in step 2 the target was reached.

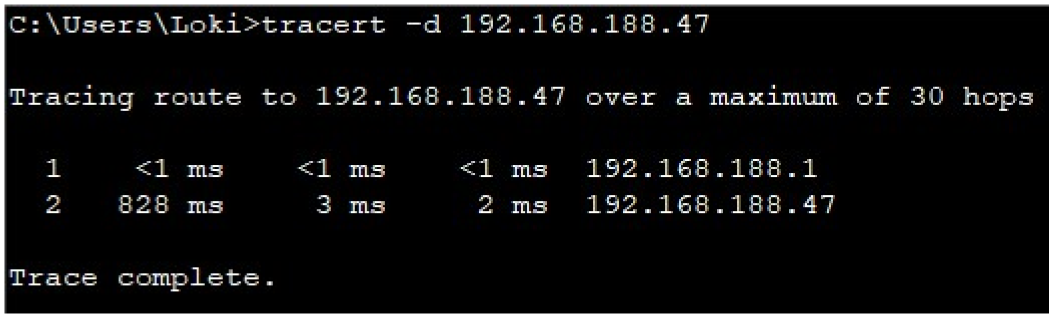
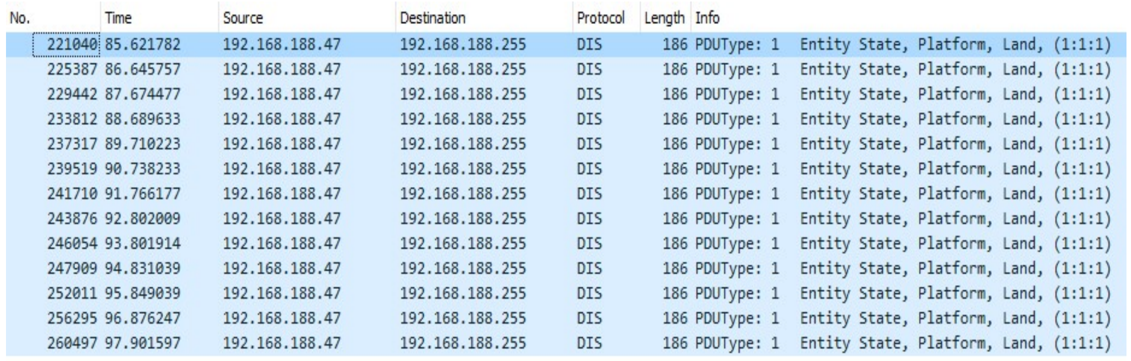

```
C:\Users\Loki>tracert -d 192.168.188.47

Tracing route to 192.168.188.47 over a maximum of 30 hops

  1    <1 ms    <1 ms    <1 ms  192.168.188.1
  2   828 ms     3 ms     2 ms  192.168.188.47

Trace complete.
```

Figure 29. Connection Test—Tracert

As mentioned before, these two basic connection tests must be done from each device to all the other devices to ensure the network setup works correctly. This is an essential basis for the next steps.

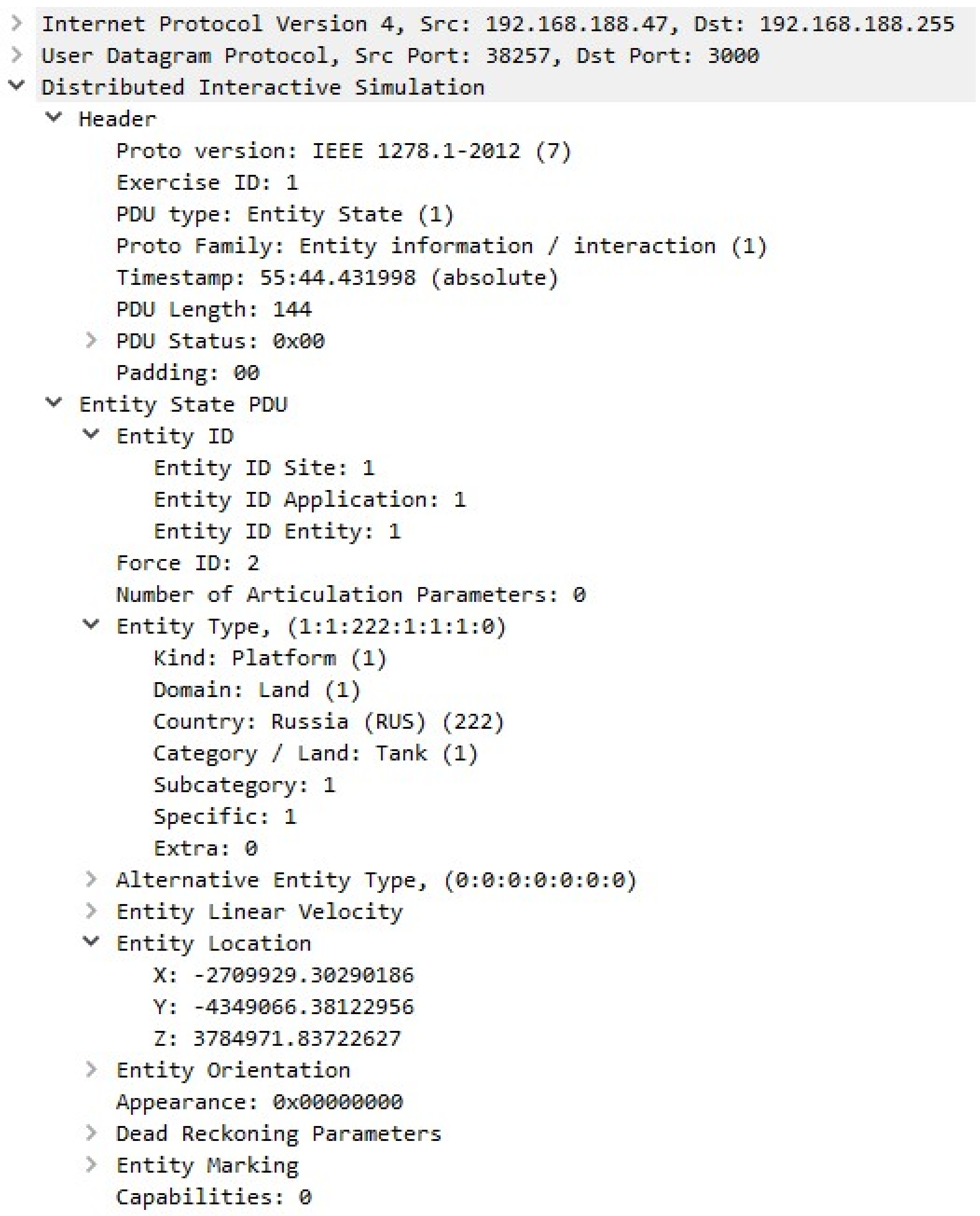
Now that we are sure that all connections are available as required, the next step is to test whether DIS packets can be sent over the network and arrive at the target computer (VR Forces). The Wireshark software is used for this. Wireshark can capture IP packets at various interfaces and make them legible for the human eye. The first step is to start the application on the mobile device. The symbol for an enemy tank was selected and the broadcast IP address 192.168.188.255 with port number 3000 was inserted. After the user presses the start button, the mobile device immediately starts sending ESPDUs into the network every second. Wireshark runs with a filter for DIS packages and captures them at the Wi-Fi interface (Figure 30).



```
No.      Time        Source          Destination      Protocol Length Info
  221040 85.621782   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  225387 86.645757   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  229442 87.674477   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  233812 88.689633   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  237317 89.710223   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  239519 90.738233   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  241710 91.766177   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  243876 92.802009   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  246054 93.801914   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  247909 94.831039   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  252011 95.849039   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  256295 96.876247   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
  260497 97.901597   192.168.188.47  192.168.188.255  DIS        186 PDUType: 1  Entity State, Platform, Land, (1:1:1)
```

Figure 30. Wireshark—Captured DIS Packages

The test proves that DIS packages were received at the destination. In the column "Time" we also can check whether the packages arrive every second. The next step is to check whether the DIS packages contain the expected data. This can be accomplished with Wireshark also simply by double-clicking on one of the listed packages (Figure 31).

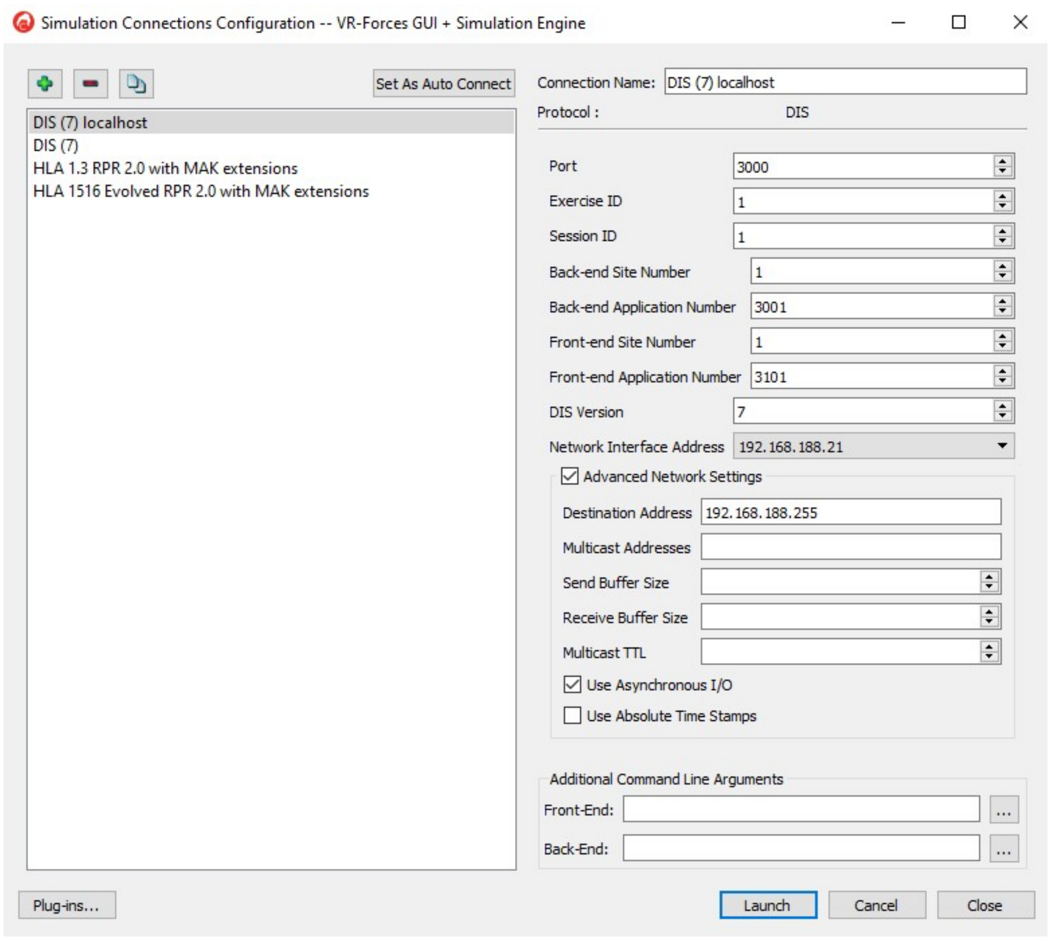


```
> Internet Protocol Version 4, Src: 192.168.188.47, Dst: 192.168.188.255
> User Datagram Protocol, Src Port: 38257, Dst Port: 3000
v Distributed Interactive Simulation
  v Header
      Proto version: IEEE 1278.1-2012 (7)
      Exercise ID: 1
      PDU type: Entity State (1)
      Proto Family: Entity information / interaction (1)
      Timestamp: 55:44.431998 (absolute)
      PDU Length: 144
    > PDU Status: 0x00
      Padding: 00
  v Entity State PDU
    v Entity ID
        Entity ID Site: 1
        Entity ID Application: 1
        Entity ID Entity: 1
      Force ID: 2
      Number of Articulation Parameters: 0
    v Entity Type, (1:1:222:1:1:1:0)
        Kind: Platform (1)
        Domain: Land (1)
        Country: Russia (RUS) (222)
        Category / Land: Tank (1)
        Subcategory: 1
        Specific: 1
        Extra: 0
    > Alternative Entity Type, (0:0:0:0:0:0:0)
    > Entity Linear Velocity
    v Entity Location
        X: -2709929.30290186
        Y: -4349066.38122956
        Z: 3784971.83722627
    > Entity Orientation
      Appearance: 0x00000000
    > Dead Reckoning Parameters
    > Entity Marking
      Capabilities: 0
```

Figure 31. Wireshark—Inside an ESPDU

As shown in Figure 31, the content could be confirmed easily. Information about the prototype version used (IEEE 1278.1-2012), entity ID, entity type (Russian land platform tank), and the location (X, Y, and Z) are transmitted. In the last step, it must be confirmed that the transferred data can be used by a C2 system (in this case, VR Forces).

To do this, the VR Forces software and a scenario are started. So that the scenario resembles as close as possible the real German C2 system, the settings are made as shown in Figure 32.


Simulation Connections Configuration -- VR-Forces GUI + Simulation Engine

Set As Auto Connect

DIS (7) localhost
DIS (7)
HLA 1.3 RPR 2.0 with MAK extensions
HLA 1516 Evolved RPR 2.0 with MAK extensions

| | |
|---|---|
| Connection Name: | DIS (7) localhost |
| Protocol : | DIS |
| Port | 3000 |
| Exercise ID | 1 |
| Session ID | 1 |
| Back-end Site Number | 1 |
| Back-end Application Number | 3001 |
| Front-end Site Number | 1 |
| Front-end Application Number | 3101 |
| DIS Version | 7 |
| Network Interface Address | 192.168.188.21 |

[✓] Advanced Network Settings

| | |
|---|---|
| Destination Address | 192.168.188.255 |
| Multicast Addresses | |
| Send Buffer Size | |
| Receive Buffer Size | |
| Multicast TTL | |

[✓] Use Asynchronous I/O
[ ] Use Absolute Time Stamps

Additional Command Line Arguments
Front-End:
Back-End:

Figure 32. VR Forces Simulation Configuration

A DIS-based simulation is chosen and the port number is set to 3000. After the scenario is started, a rudimentary map is shown. Scrolling to the area of the Monterey Peninsula, we can identify the coastline. Unfortunately, the resolution of the map material is very low, and no details can be displayed, but for this kind of prototype testing that is not relevant. The next step is to start the application and select the military symbol of a friendly tank. After checking the destination IP address as well as the port number, we press the “start” button. In less than one second, the symbol appears on the map of VR

Forces at the expected location. We can check this with the shown coordinate system. The symbol is also displayed with the label “Leopard 2A6” tank, which is exactly the selected type (Figure 33).

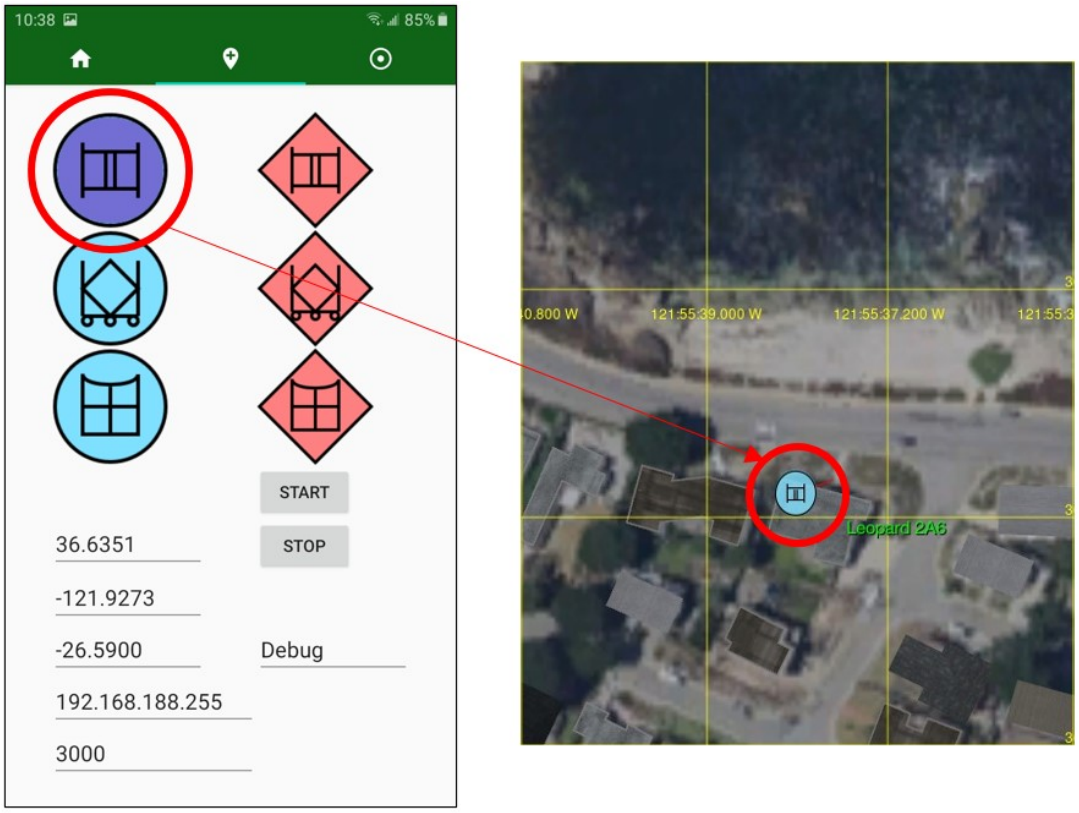
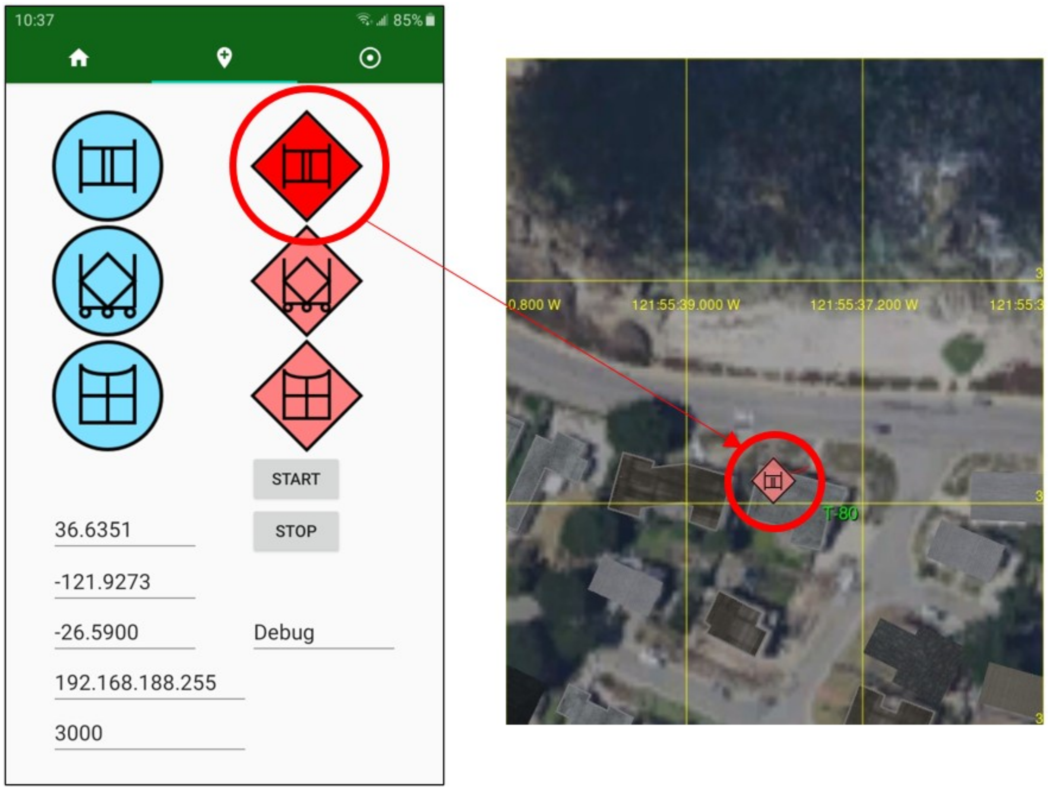

36.6351
-121.9273
-26.5900
192.168.188.255
3000
Debug

Figure 33. Test Application—VR Forces Friendly Tank

If the mobile device is moved, the location of the symbol changes also. Because of the resolution of the map used, this could not be shown in this first prototype test, but it could be seen in Wireshark.

The next step is to change the selected military symbol from friendly tank to an enemy tank. The displayed symbol at VR Forces changes immediately (max. one second after activating). Also, the label for that symbol changes to “T-80” (Figure 34). This test was performed for all six different military symbols and at different places within the range of the Wi-Fi network used.



36.6351

-121.9273

-26.5900

192.168.188.255

3000

Figure 34. Test Application—VR Forces Enemy Tank

## B. MOBILE HIT ACTOR

While the previous section dealt with the development of an entity simulation that enables the display of any unit or any vehicle in a training environment, this section deals with the development of a mobile actuator that can react to virtual events in real life. Specifically, it is examined whether a mobile and flexible platform developed by simple means can react to virtual fire (AGDUS) events with sound and/or other effects.

The requirements for this device are having a small size, autonomous operating time of at least six hours, and the potential to be connected to the communication device in the vehicle. To achieve this, a compact computer platform with low energy consumption and maximum connectivity was selected. The minicomputer "Raspberry Pi" was chosen. This is a single-board computer about the size of a credit card and is competitive with a small PC in terms of performance. Its diverse interfaces (Ethernet, Wi-Fi, Bluetooth) and special input and output ports as well as its low price make it perfect for this project. An average power consumption of approximately 2 watts also allows it to temporarily operate independently, e.g., with the help of a power bank.

### 1. Setup

The setup for the “mobile hit actor (MHA)” remains the same as described in the project “mobile entity simulator” and is not repeated here. The Raspberry Pi and its accessories are now also required for this setup.

#### *a. Raspberry Pi*

The following paragraphs describe the hardware and software components of the Raspberry Pi used in this setup.

##### (1) Hardware

The minicomputer Raspberry Pi has been produced and sold in different versions since 2006. At the time of this work, version “4 Model B” is current. Nonetheless, a “3 Model B+,” which has been available since 2018, is used for this setup due to its availability. The parameters of this version are listed in Table 6.

Table 6. Hardware Specifications for the Raspberry Pi

| | **Value** |
|---|---|
| **CPU** | Arm Cortex-A53 CPU @ 1.2 GHz |
| **GPU** | Broadcom Dual Core Video Core IV 400 MHz |
| **RAM** | 1 GB |
| **System Type** | 64-bit OS, x64-based processor |
| **Ethernet** | Gigabit Ethernet |
| **Wi-Fi** | 2.4 + 5 GHz |
| **Bluetooth** | 4.1 |
| **Audio** | 3.5 mm |
| **GPIO** | 48 pins |
| **USB** | 2.0 |

To make this device fully capable for the purpose of this research, additional parts were needed. Because there is no audio output device within the minicomputer, an external speaker was added and connected via a 3.5 mm sound cable. Also, a power bank with 50,000 mAh was used to make the whole setup independent and mobile. It ensures a runtime of approximately ten hours. Figure 35 shows this setup.

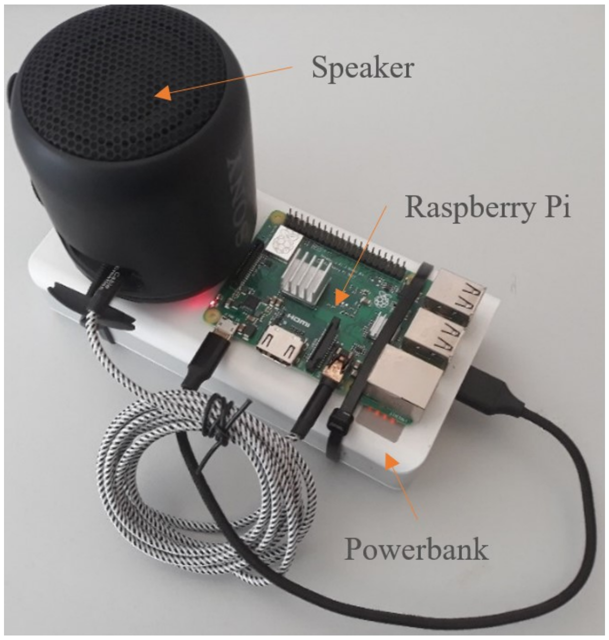


Figure 35. Mobile Hit Actor Hardware Setup

(2) Software

Various OS are available for the Raspberry Pi. We could choose from Linux versions, a Microsoft Windows version, and an available Android version. Unfortunately, not all OS supports all interfaces and capabilities of the hardware. Therefore, a Linux OS that supports all the needed capabilities was chosen. For the programming language, the Java is used on this platform. The complete software setup is shown in Table 7.

Table 7. Software Specifications for the Raspberry Pi

| | Value |
|---|---|
| **OS** | Raspberry Pi OS “desktop” Kernel 5.4 |
| **Java** | OpenJDK version 11 |
| **DIS** | openDIS 7 library |

The installation of an OS on a Raspberry Pi is not comparable to the installation on an ordinary computer. Since this minicomputer has no hard drive or the like, the entire software—including the operating system—must be installed on a memory card. This is usually done with the following steps:

1. Download the OS ISO image (*.iso format).
2. Write the ISO image onto the memory card:
    - Windows: Tools like Win32 Disk Imager can be used.
    - Linux: Use command “dd bs=__ if=/__ of=__,” where bs is the block size of the image, if is the path to the input file, and of is the path to the memory card.
3. Insert the memory card into the Raspberry Pi.
4. Boot up the Raspberry Pi from the memory card.
5. Install all additional software with the chosen OS tools.

### *b. Network Setup*

For developing and testing, the described device must work together in a network such as the one shown in Figure 44. In addition to those development steps, the Raspberry Pi must be inserted (Figure 36). The software is developed at the notebook and deployed via a secured shell (ssh) connection to the Raspberry. The MHA can receive DIS packages over the network either from the C2 system or from the application running on the mobile device in this setup.

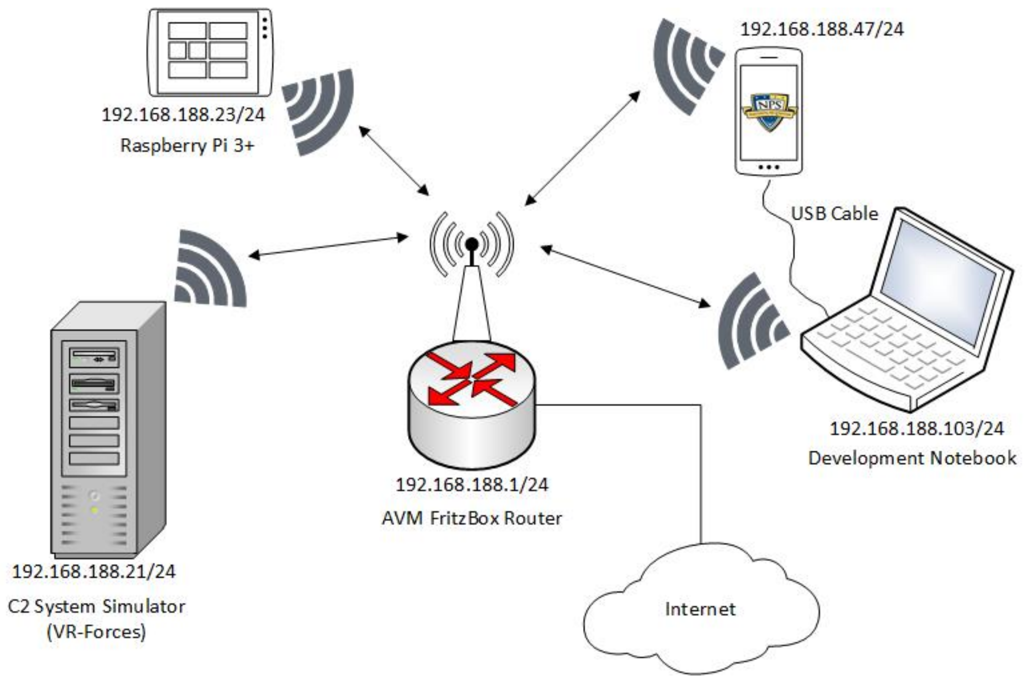


Figure 36. Mobile Hit Actor Development Setup

### 2. Prototype Development

The development process for the MHA is divided into two parts. First is the development of the hit actor itself. This software part will run on the Raspberry Pi. It must be programmed in Java, as the system architecture does not allow Kotlin code. The main task of this program is to track DIS traffic in the network and to respond appropriately to specific packets. In particular, about the program must detect detonation PDUs that are directed to the entity ID of the vehicle to whose ComSys the MHA is connected. When the software receives a corresponding PDU, a short explosion sound should be played over the loudspeaker for demonstration purposes.

Second, a test program must be developed that can send targeted detonation PDUs to test the function of the MHA. For the sake of simplicity, this test program is programmed in Kotlin and becomes part of the "Mobile Entity Simulator (MES)" application that has already been produced. This enables mobile use and tests can be carried out in flexible locations.

### *a. MHA Development Process*

This application does not need a graphical user interface in the prototype version. No display is connected, and no inputs must be done. The application should start automatically after power up of the hardware and should play the detonation sound once to alert the user that the system is ready. Figure 37 shows the program flow in detail.

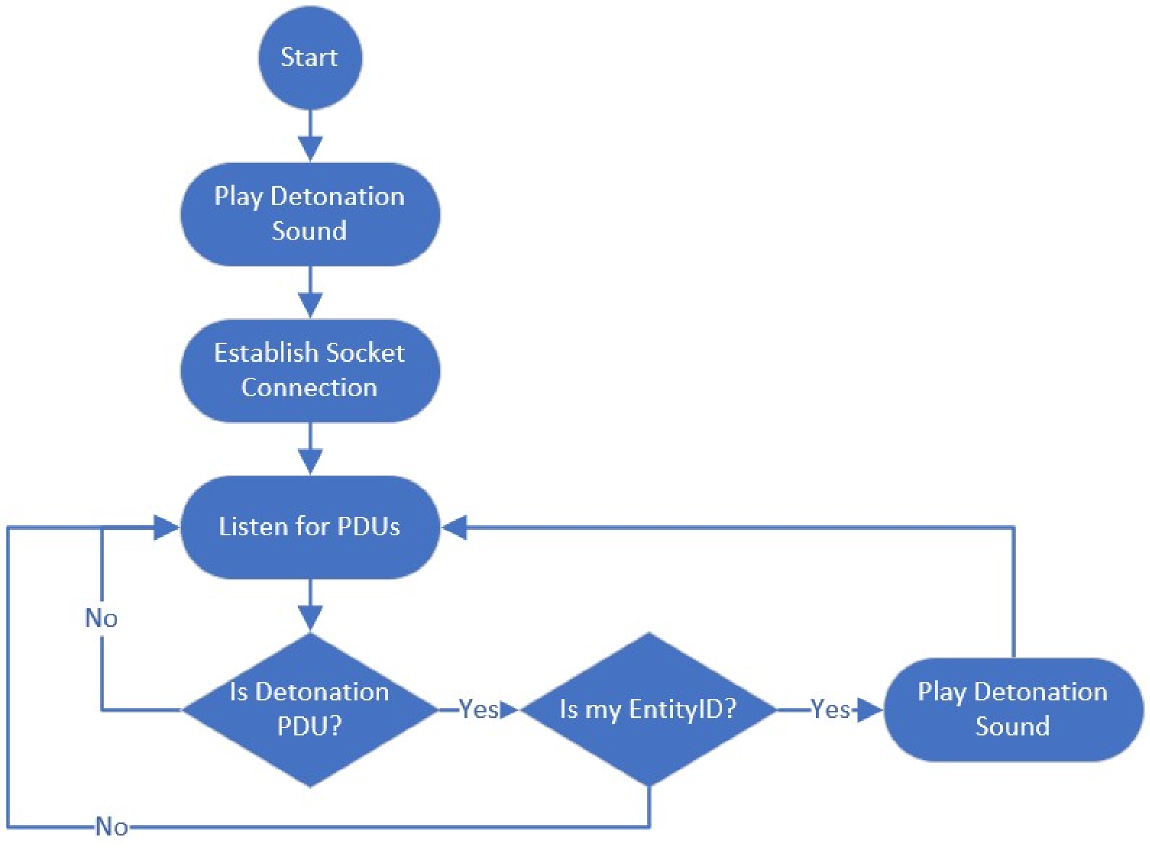


Figure 37. Flow Chart of Mobile Hit Actor

After the MHA has started up, a detonation sound should be played (as just described). A multicast socket connection is then created and the software listens to the selected port for DIS packets. If a DIS packet is recognized, it is checked to determine whether it is a detonation PDU. If it is not, the listening continues. But if it is a detonation PDU, it is checked to determine whether the entity ID of the target in the packet matches the entity ID of the vehicle on which the MHA is located. If this is the case, the detonation sound is played again; otherwise, further DIS packets are listened for. The program runs for as long as the Raspberry Pi is supplied with power. The development of the source code is straightforward, and a detailed description of each individual step can be dispensed with at this point. The entire source code can be found in Appendix B.

The produced source code must be packed together with the libraries used and the sound file and transferred from the development PC to the Raspberry Pi. NetBeans supports this process. Eight steps are needed to accomplish this and are described here:

1. Right-click on the Project name.
2. Select "Properties."
3. Click "Packaging."
4. Check "Build JAR after Compiling" and "Copy Dependent Libraries."
5. Check "Compress JAR File."
6. Click OK to accept changes.
7. Right-click on a Project name.
8. Select "Build" or "Clean and Build."

NetBeans generates a *.jar file in a generated sub-folder "dist." This *.jar file must be placed in the file system of the Raspberry Pi. This could be done by using an SSH connection or simply with a USB memory stick. After the bytecode has been ported to the Raspberry, it should start automatically when the system starts up. A script is created for this purpose, which is executed when the system is started. The content of the script is given here:

```
#!/bin/bash
cd /…/  //path to folder where the script is
java -jar Raspi.jar
```

The script must be executable. To ensure that it is, the script file parameters must be changed. This is done by the following command:

```
chmod u+x /path/to/script/scriptname.sh
```

The “chmod” command is a Linux command that allows for changing the permissions of a file,  while “u+x” means that every user can execute this script. In order to run the script automatically when the system is started, it must be moved to a special folder within the Linux operating system. This order in which this is done can be different for different derivatives. In the Raspberry Pi OS used here, it is the folder:

```
/etc/init.d
```

### *b. MHA Test Application*

As described before, a test program is needed to check the functionality of the developed MHA. To keep the test program simple, it is inserted into the existing MES application. A rudimentary graphical user interface is needed as shown in Figure 38. The input fields for the IP address and port number are located in section 1. Section 2 displays a simple button that should send one detonation PDU to the stated address each time it is pressed.

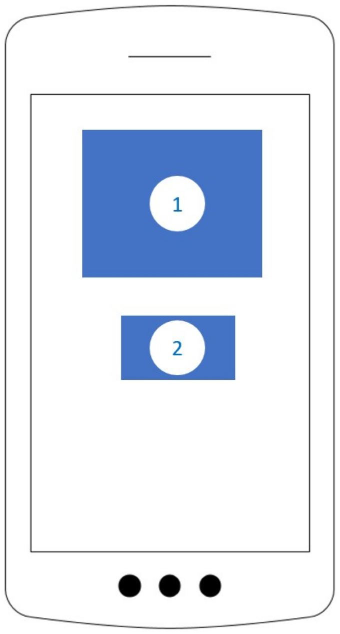


Figure 38. Design of Human Interface for Mobile Hit Actor Test Application

The flow chart for this testing application is straightforward and described in Figure 39. Five sequential use cases operate in the order shown. After the IP address and port number are inserted, the user can initiate the process of sending a detonation PDU by pressing a button. This action starts the routine with three steps: establish a socket connection, build detonation PDU, and send the created package. These steps could be repeated with or without changing the IP address and/or port number.

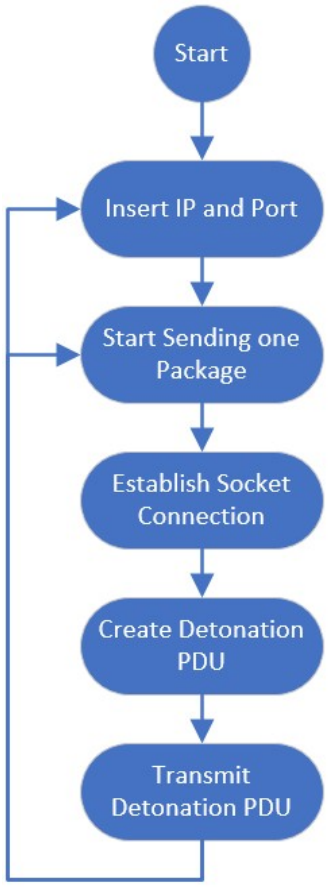
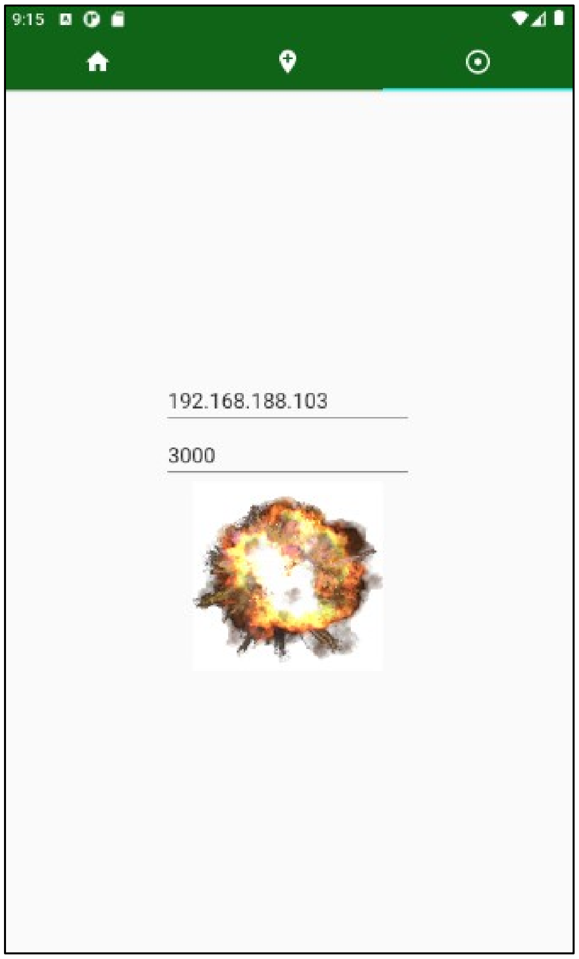


Figure 39. Flow Chart of Mobile Hit Actor Test Application

All needed use cases are described in detail in section A where the development of the MES is discussed. The only slight difference here is the "create detonation PDU." After a new DetonationPDU object is initiated, a time stamp is created and added to this object. For the given purpose, no location data or other specific information is needed except the entity ID, because the MHA will check whether this ID is responsible for the detonation.

The following short code sequence shows this part. The whole Kotlin code is provided in section F of Appendix B. To generate the layout for this little test application, fragment_send_detonation.xml was created in the layout folder of the Android Studio project. Again, this was straightforward, and the code is also provided in section G of Appendix B for the sake of completeness. The resulting output is shown in Figure 40.

```
// creating the detonation PDU

    var detPDU = DetonationPdu()
    var timeStamp = DisTime().disAbsoluteTimestamp
    detPDU.timestamp = timeStamp
    // setting the target EntityID
    detPDU.setTargetEntityID(EntityID().setEntityID(321.toShort()))
```



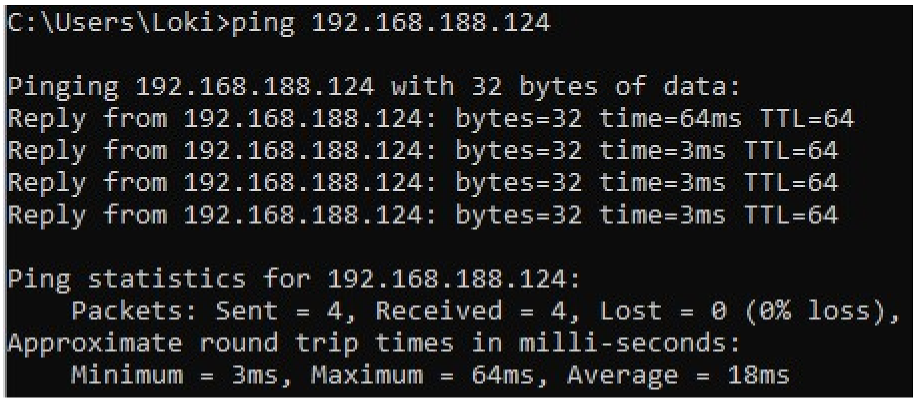
192.168.188.103

3000

Figure 40. Screenshot of Mobile Hit Actor Test Application

### 3. Testing and Demonstration

Testing the mobile hit actor is similar to testing the MES. Again, the first step is to make sure the network connection is working. This test was particularly important in this case, as it showed that the IP address of the MHA changed from 192.168.188.23 to 192.168.188.124 compared to the initial structure (Figure 36). This can be explained by the fact that IP addresses are assigned to end devices in the test setup using the Dynamic Host Configuration Protocol (DHCP) by the AVR router. Several weeks passed between the initial setup of the construction and the testing. This result shows how important these supposedly simple tests are. After evaluation in the setup of the router, the new IP was determined and the connection between the devices was confirmed by means of a "ping" test (Figure 41).

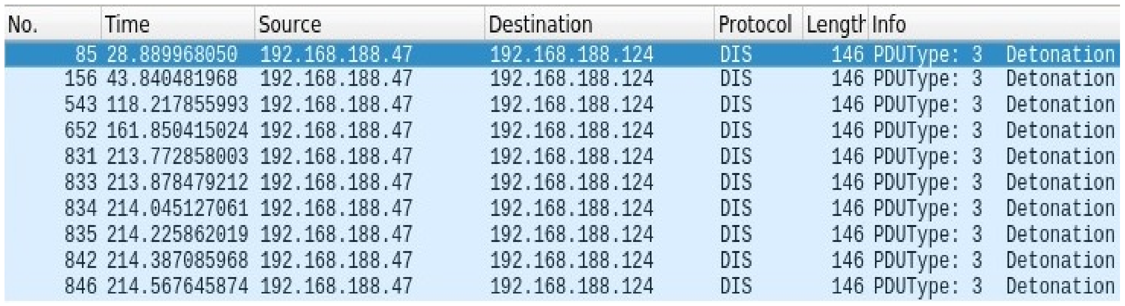


```
C:\Users\Loki>ping 192.168.188.124

Pinging 192.168.188.124 with 32 bytes of data:
Reply from 192.168.188.124: bytes=32 time=64ms TTL=64
Reply from 192.168.188.124: bytes=32 time=3ms TTL=64
Reply from 192.168.188.124: bytes=32 time=3ms TTL=64
Reply from 192.168.188.124: bytes=32 time=3ms TTL=64

Ping statistics for 192.168.188.124:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 3ms, Maximum = 64ms, Average = 18ms
```

Figure 41. MHA Connection Test

After a stable connection is confirmed, the next test examines whether DIS packets that are sent by the MHA test application also arrive at the MHA. To check this, some additional measures are necessary. As described before, the MHA has a small design and can be operated independently (but within a network). The fact that the hardware does not have any input or output devices means that additional equipment must be connected for the period in which the tests are to be carried out. Specifically, a monitor is connected via the HDMI connection on the Raspberry Pi board and a keyboard and mouse are connected via the USB connections.

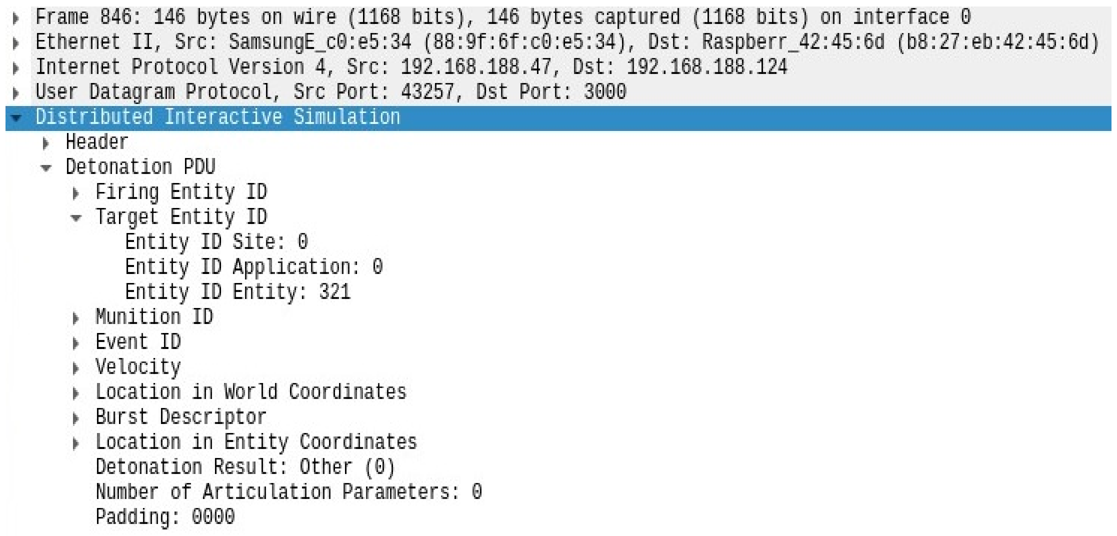
As with the MES, the Wireshark software can now be used to record data traffic via special interfaces. In the specific test scenario, the traffic is captured via the internal Wi-Fi chip of the Raspberry Pi. For reasons of clarity, the filter was set to "DIS." Now the MHA test application is started and the IP address 192.168.188.124 is entered. Each time the detonation button is clicked, a packet is sent to the MHA and registered and recorded by Wireshark. Figure 42 shows the result of this output.



```
No.  Time           Source          Destination      Protocol Length Info
  85 28.889968050   192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 156 43.840481968   192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 543 118.217855993  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 652 161.850415024  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 831 213.772858003  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 833 213.878479212  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 834 214.045127061  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 835 214.225862019  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 842 214.387085968  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
 846 214.567645874  192.168.188.47  192.168.188.124  DIS       146 PDUType: 3   Detonation
```

Figure 42. MHA Wireshark—Captured DIS Packages

The next step is to check whether the packages are being delivered and received in the correct format. In the test output beforehand, it can be verified that the DIS packs are packets of the "detonation" type (PDUType 3). A double-click on one of the received packets opens the detailed view and confirms the first impression (Figure 43). The packets sent in UDP are detonation PDUs. A closer look makes it clear that under the header "Target Entity ID" the value of the attribute "Entity ID Entity" is 321—precisely the value that was stored in the MHA test application in the code (see the snippet of code shown previously in Figure 40).



```
▸ Frame 846: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits) on interface 0
▸ Ethernet II, Src: SamsungE_c0:e5:34 (88:9f:6f:c0:e5:34), Dst: Raspberr_42:45:6d (b8:27:eb:42:45:6d)
▸ Internet Protocol Version 4, Src: 192.168.188.47, Dst: 192.168.188.124
▸ User Datagram Protocol, Src Port: 43257, Dst Port: 3000
▾ Distributed Interactive Simulation
  ▸ Header
  ▾ Detonation PDU
    ▸ Firing Entity ID
    ▾ Target Entity ID
        Entity ID Site: 0
        Entity ID Application: 0
        Entity ID Entity: 321
    ▸ Munition ID
    ▸ Event ID
    ▸ Velocity
    ▸ Location in World Coordinates
    ▸ Burst Descriptor
    ▸ Location in Entity Coordinates
      Detonation Result: Other (0)
      Number of Articulation Parameters: 0
      Padding: 0000
```

Figure 43. MHA Wireshark—Inside Detonation PDU

The last and most important step is to test whether the sound can also be played after the detonation PDU package has been received. To do this, the connected loudspeaker is activated and connected to the Raspberry Pi using a cable jack. Every click on the detonation button is immediately acknowledged by playing the detonation sound. Obviously, this cannot be documented with a screenshot—but it works great.

THIS PAGE INTENTIONALLY LEFT BLANK

# V. CONCLUSIONS AND RECOMMENDATIONS

## A. CONCLUSIONS

The work in this thesis has shown that small mobile devices can be a useful addition to live training sessions to significantly improve the simulation of reality. More realistic training then can lead to a better training result.

As this thesis has shown, the development of custom software systems for existing hardware is feasible with relatively little effort. Vehicles participating in an exercise can be represented with any military symbols in the C2 system and virtual hits can also effectively simulate a realistic impact.

During the thesis, the following research questions set out in the introduction were processed and answered.

- **Question 1:** “What wireless communications systems are available to network Android devices, and what are the advantages and disadvantages of each for the proposed application?”

- **Answer:** Since a mobile, reliable connection to an existing communication system within a vehicle is required, only the connection options of this ComSys came into question. Based on the analysis of this selection, the use of the Wi-Fi interface was recommended. In addition, a connection via 5G was also considered. Although this is not an option today due to the lack of appropriate technical equipment technical equipment, it will certainly be a possible improvement in the future.

- **Question 2:** “What simulation interoperability protocol or architecture should be used to pass this tactical information and why (i.e., what potential architectures have to be researched)?”

- **Answer:** This question was relatively easy to answer. Typically, HLA is preferred as the architecture because of the flexibility it provides and especially since all the important states in the Western world have agreed on it as the standard. However, DIS was preferable in this specific case since the German C2 system this research intends to address only "speaks" DIS. So, this protocol was used for the software development.

- **Question 3:** "Are there significant differences in operating the DIS protocol versus the HLA when a number of devices are all streaming to an existing simulation system for visualization?"
- **Answer:** This could not be answered satisfactorily in the course of this work. Since the definition of DIS in the German C2 system left only one development path, the result did not go beyond the theoretical differences between HLA and DIS. A more detailed answer to this question can be part of a future work.
- **Question 4:** "Once this data is broadcast as desired, how will it be received and bridged into the tactical system used to integrate into the training environment?"
- **Answer:** Here, too, there were few possibilities due to the test construction. If the C2 system and the mobile device with the application are in the same network, the data can be transmitted via multicast without any problems. This was particularly evident in the MES prototype. If one of the two devices is on a different network, the router hardware used determines whether multicast via VPN is supported. While the AVR router can support this function, the hardware used at the Naval Postgraduate School does not at the time of this work.
- **Question 5:** "Is it possible to develop a prototype on a mobile (Android) device that considers and confirms the previous results? What programming language should be used (Kotlin or Java) and why?"
- **Answer:** "It depends." If an application is developed for an Android device, Kotlin seems to be the better programming language to use. In

other cases, as shown in the example of the MHA, however, Java can be used.

## B. RECOMMENDATIONS FOR FUTURE WORK

Above all, the MHA can be significantly further developed. The 48 GPIO pins make it possible to deliver different reactions to different hits. As shown with just the one port used in the thesis, the developed system can play a sound when hit. Consequently, if the additional input / output ports are used, functions such as triggering smoke, blocking individual hardware functions of the combat vehicle, or other functions can be carried out.

The systems developed, which included the MES and the MHA, still must undergo further experimentation to show that they can actually improve the effectiveness of live training significantly. This is to be carried out as part of a study in 2022 on a German training site.

Furthermore, the answer to the third research question was not satisfactory. The developing systems must "understand and speak" both DIS and HLA. Once this has been achieved, comparative tests should be carried out to determine which protocol or which architecture has what advantages and disadvantages in a simulation environment.

THIS PAGE INTENTIONALLY LEFT BLANK

# APPENDIX A. DEVELOPMENT SETUP

## A. SETUP

To make my work repeatable I describe the setup of the developing and testing environment in this section.

### 1. Development Computer

Description of the used hard- and software of my laptop for the developing process.

#### *a. Hardware*

A Lenovo ideapad 330S notebook is used for the development process. The specifications for this computer can be found in Table 8. A large amount of RAM and a powerful GPU are needed to run the software development kit and the mobile device simulators. Also, USB connectors (versions 2.0 and 3.0) are needed.

Table 8. Hardware Specifications of Development Computer

| | **Value** |
|---|---|
| **CPU** | Intel Core i7-8550U CPU @ 2.0 GHz |
| **GPU** | Nvidia GeForce GTX 1050 with 14GB RAM |
| **RAM** | 20.0 GB |
| **System Type** | 64-bit OS, x64-based processor |

#### *b. Software*

In this work, a Windows OS is used with Android Studio as the software development kit (SDK). Detailed software specifications can be found in Table 9.

Table 9. Software Specifications of Development Computer

| | Value |
|---|---|
| **OS** | Windows 10 Home version 1909 build 18363.1279 |
| **SDK Android** | Android Studio version 4.0.1 build 193.6911.18; Runtime version: 1.8.0_242 amd64; VM: OpenJDK 64-Bit Server VM by JetBrains |
| **SDK Java** | Apache NetBeans IDE 12.0 |
| **Java** | OpenJDK version 14.0.1 |
| **Kotlin** | 1.4.21-release-Studio 4.0-1 |
| **DIS** | openDIS 7 library |

### 2. Mobile Device

Description of the used hard- and software of the mobile device used for tests.

#### *a. Hardware*

To test the new application a mobile device is needed. In a first step a common smart phone with an Android OS is used. Important for the development process is the possibility to connect the device with the development computer. Android Studio can use a cable connection (USB) to deploy code to the mobile device directly and execute the code. Detailed specifications for the mobile device hardware can be found in Table 10.

Table 10. Hardware Specifications for Mobile Device

| | Value |
|---|---|
| **Device** | Samsung Galaxy J7 |
| **Model** | SM-J737U |
| **Display** | 5,5 inches; 720 x 1280 Pixel |
| **CPU** | Qualcomm Snapdragon 615 |
| **Memory** | 16 GB |
| **Connectivity** | NFC, Bluetooth, Wi-Fi, USB, LTE |

#### *b. Software*

No special software is needed. The Samsung Galaxy J7 runs with Android 9 “Pie” and the patch level from December 1, 2020. The implemented kernel version is 3.1891-16371010 and Knox 3.3.

One important step ensures this device can be used for development and testing. The phone must be switched into the development mode. This status enables Android Studio on the development computer to deploy a new application version over the USB to the mobile device and run it there. To switch the Samsung J7 into development mode a few steps that must be performed are described here:

1. Go to “Settings” (by clicking on the gearwheel icon).
2. Scroll down to the tab “About phone” and select it.
3. Select “Software information.”
4. Tap “Build number” seven times.
5. Go back to “Settings.”
6. A new tab “Developer options” appears.
7. Tap “Developer options.”
8. Enable developer mode (by switching “On”).
9. Enable USB debugging.
10. Disable verify apps over USB.
11. Leave everything else on default.

### 3. C2 System Simulator (VR Forces)

Description of the used hard- and software for simulation a German C2 System.

### *a. Hardware*

As described before, VR Forces is used to simulate a part (force tracking) of a real C2 system. A powerful machine with a specific graphics card is needed to support the C2 system simulator. The specifications of the machine used can be found in Table 11.

Table 11. Hardware Specifications for C2 Simulator

| | **Value** |
|---|---|
| **CPU** | AMD Ryzen 7 2700X Eight Core Processor @ 3.7 GHz |
| **GPU** | Nvidia GeForce GTX 1070 with 16GB RAM |
| **RAM** | 16.0 GB |
| **System Type** | 64-bit OS, x64-based processor |

### *b. Software*

To run VR Forces for this purpose, a Windows OS is needed. The detailed software specifications for this machine can be found in Table 12.

Table 12. Software Specifications of C2 Simulator

| | **Value** |
|---|---|
| **OS** | Windows 10 Pro version 2004 build 19041.804 |
| **VR Forces** | Version 4.8, 64-Bit |
| **Wireshark** | Version 3.4.3 64-Bit |

Wireshark is an open-source tool used to capture and analyze data traffic. It can dump data packages on every interface of the computer (Ethernet, Wi-Fi, Bluetooth, USB). Various filters and settings allow developers to evaluate and troubleshoot incidents in a network.

### 4. Network Setup

For development and testing of the three devices described previously, they must work together in a network. Figure 44 shows the setup used during the development process. A typical development and testing process in an early phase can be described in these four steps:

1. Source code is developed on the development computer (Android Studio).
2. Compiled and packed code is deployed to the mobile device (Samsung Galaxy J7) via USB cable.
3. The Running application sends data using Wi-Fi.
4. The C2 system simulator receives data.

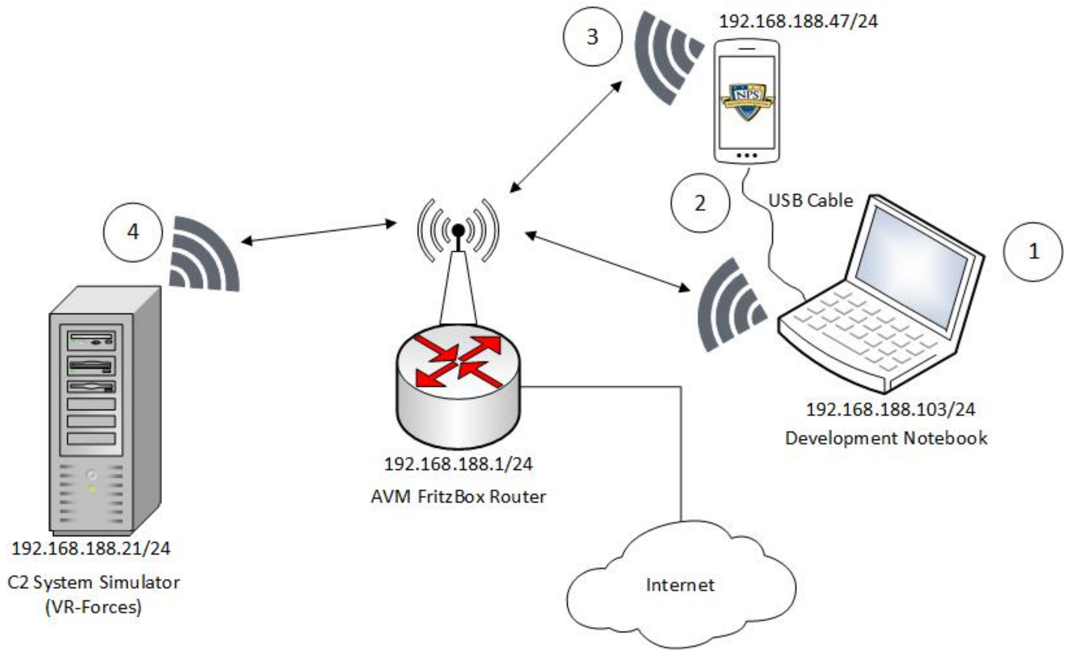
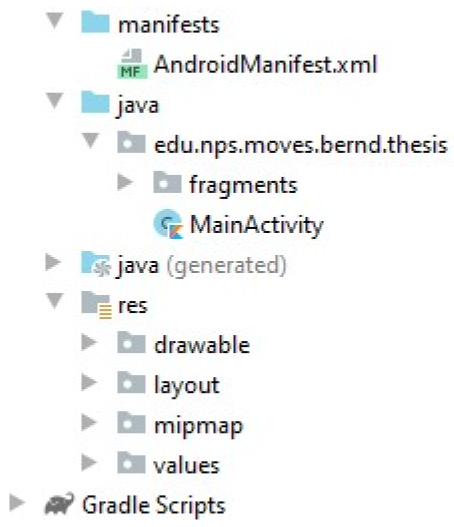


Figure 44. Development Network Setup

The internal Wi-Fi network is a Class C with the network address 192.168.188.0 and the subnet mask 255.255.255.0. IP addresses are deployed by an AVM FritzBox router (Model: FRITZ!Box Fon WLAN 7390, FRITZ!OS: 06.86) using DHCP.

## B. ANDROID STUDIO

Developing an application in Java or Kotlin for mobile Android-driven devices differs in some points from other systems. This section briefly describes how Android Studio manages the files in an internal structure and what purpose it serves. Figure 45 gives an overview of how this is done. Detailed information can be found in the book *Head First Kotlin—A Brain-Friendly Guide* (Griffiths & Griffiths, 2019). It provides a very good overview for people who have never coded with Android Studio for mobile devices.



```
▼ manifests
    AndroidManifest.xml
▼ java
  ▼ edu.nps.moves.bernd.thesis
    ▶ fragments
      MainActivity
▶ java (generated)
▼ res
  ▶ drawable
  ▶ layout
  ▶ mipmap
  ▶ values
▶ Gradle Scripts
```

Figure 45. Typical Project Structure in Android Studio

As shown in Figure 45, a simple and basic project exists out of four main folders plus one other: “manifest,” “java,” “java(generated),” “res,” and “Gradle Scripts.”

### 1. The “manifest” Folder

This folder contains the AndroidManifest.xml file. This file is a kind of intermediator between the application and the Android OS on the device. It contains metadata about the Android version for which this application is suitable, information about the status package for Kotlin files, and the application’s access rights to hardware components of the device (GPS, Wi-Fi, etc.). In addition, basic information such as the icon used for the application and the style theme used are defined here.

### 2. The “java” and “java (generated)” Folders

All Java and/or Kotlin source code produced during the programming work is in this folder. If a new project is created, a MainActivity file is also automatically generated. Depending on whether Kotlin or Java was selected as the programming language, this file ends with .java or .kt. This file is located at the top level of the folder structure and is a prerequisite for every project. The rest of the source code can be arranged in a tree structure (edu.nps.moves.bernd. ...). As with most other programming languages, this method is always advisable when creating a complex project.

Compared to “java,” the “java (generated)” folder contains only the files which, as the name suggests, are generated by the SDK. Normally, a developer does not have to deal with this part of the project.

### 3. The “res” Folder

The resource folder contains all the non-code sources needed and used within the project. Different types of resources are structured in different folders. The “drawable” folder includes all images and icons. The “layout” folder contains the *.xml files that describe the layout of the application. Within the “midmap” folder all the needed images and icons are defined and stored in different resolutions (hdpi, mdpi, xhdpi). Finally, “values” includes different *.xml files that define the look and feel of the application. Dimensions, styles, and colors of the application can be changed here. An important file is strings.xml. All strings used in the application can be defined here. This could be very helpful for a developer who must provide the program for different countries with different languages.

### 4. The “gradle scripts” Folder

In this context, gradle means “automated build system.” Mainly all the files stored here contain the information required by Android Studio to build the project. The additional modules and plugins that are part of the project are also defined here.

THIS PAGE INTENTIONALLY LEFT BLANK

# APPENDIX B. SOURCE CODE

## A. ENTITY STATE SIMULATOR—FRAGMENT_SEND_POSITIO.XML

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".fragments.SendPositionFragment">


    <GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:orientation="horizontal">

        <ToggleButton
            android:id="@+id/button_NotPressed_KPz_blue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:background="@drawable/pzmedblue_button"
            android:textOff=""
            android:textOn="" />

        <ToggleButton
            android:id="@+id/button_NotPressed_SPz_blue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="1"
            android:layout_column="0"
            android:background="@drawable/spzblue_button"
            android:textOff=""
            android:textOn="" />

        <ToggleButton
            android:id="@+id/button_NotPressed_San_blue"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="2"
            android:layout_column="0"
            android:background="@drawable/sanblue_button"
            android:textOff=""
            android:textOn="" />

        <ToggleButton
            android:id="@+id/button_NotPressed_KPz_red"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="0"
            android:layout_column="1"
            android:background="@drawable/pzmedred_button"
            android:textOff=""
            android:textOn="" />

        <ToggleButton
            android:id="@+id/button_NotPressed_SPz_red"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="1"
            android:layout_column="1"
            android:background="@drawable/spzred_button"
            android:textOff=""
            android:textOn="" />

        <ToggleButton
            android:id="@+id/button_NotPressed_San_red"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="2"
            android:layout_column="1"
            android:background="@drawable/sanred_button"
            android:textOff=""
            android:textOn="" />

        <Button
            android:id="@+id/startButton"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_row="3"
            android:layout_column="1"
            android:text="Start" />

        <Button
            android:id="@+id/stopButton"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_row="4"
            android:layout_column="1"
            android:text="Stop" />

        <EditText
            android:id="@+id/ipTextField"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="9"
            android:layout_column="0"
            android:ems="8"
            android:inputType="textPersonName"
            android:text="192.168.188.255" />

        <EditText
            android:id="@+id/portTextField"
            android:layout_width="wrap_content"
```

```
            android:layout_height=“wrap_content”
            android:layout_row=“10”
            android:layout_column=“0”
            android:ems=“8”
            android:inputType=“number”
            android:text=“3000” />

        <EditText
            android:id=“@+id/latitude_value”
            android:layout_width=“wrap_content”
            android:layout_height=“wrap_content”
            android:layout_row=“4”
            android:layout_column=“0”
            android:ems=“6”
            android:inputType=“textPersonName”
            android:text=“latitude” />

        <EditText
            android:id=“@+id/longitude_value”
            android:layout_width=“wrap_content”
            android:layout_height=“wrap_content”
            android:layout_row=“6”
            android:layout_column=“0”
            android:ems=“6”
            android:inputType=“textPersonName”
            android:text=“longitude” />

        <EditText
            android:id=“@+id/altitude_value”
            android:layout_width=“match_parent”
            android:layout_height=“wrap_content”
            android:layout_row=“7”
            android:layout_column=“0”
            android:ems=“6”
            android:inputType=“textPersonName”
            android:text=“altitude” />

        <EditText
            android:id=“@+id/output”
            android:layout_width=“match_parent”
            android:layout_height=“wrap_content”
            android:layout_row=“7”
            android:layout_column=“1”
            android:ems=“6”
            android:inputType=“textPersonName”
            android:text=“Debug” />

    </GridLayout>
</FrameLayout>
```

## B. ENTITY STATE SIMULATOR—MAINACTIVITY.KT

```
package edu.nps.moves.bernd.thesis

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import edu.nps.moves.bernd.thesis.fragments.HomeFragment
import edu.nps.moves.bernd.thesis.fragments.SendDetonationFragment
import edu.nps.moves.bernd.thesis.fragments.SendPositionFragment
//import edu.nps.moves.bernd.thesis.fragments.SettingsFragment
import edu.nps.moves.bernd.thesis.fragments.adapters.ViewPagerAdapter
import kotlinx.android.synthetic.main.activity_main.*

/**
 * MainActivity
 * Title:  Thesis Prototype
 * Author: LTC Bernd Weissenberger
 * Date:   02/19/2021
 *
 * Defines the basic layout of the app.
 */

class MainActivity : AppCompatActivity() {

    /**
     * Called once after open the app
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setUpTabs()
    }

    /**
     * Generates the fragments.
     * Actual version: 3 fragments (Fragment "settings" is deactivated)
     */
    private fun setUpTabs(){
        val adapter = ViewPagerAdapter(supportFragmentManager)
        adapter.addFragment(HomeFragment(), ““)
//        adapter.addFragment(SettingsFragment(), ““)
        adapter.addFragment(SendPositionFragment(), ““)
        adapter.addFragment(SendDetonationFragment(), ““)
        viewPager.adapter = adapter;
        tabs.setupWithViewPager(viewPager)

        tabs.getTabAt(0)!!.setIcon(R.drawable.ic_baseline_home_24)
//        tabs.getTabAt(1)!!.setIcon(R.drawable.ic_baseline_settings_24)
        tabs.getTabAt(1)!!.setIcon(R.drawable.ic_baseline_add_location_24)
        tabs.getTabAt(2)!!.setIcon(R.drawable.ic_baseline_adjust_24)
    }
}
```

## C. ENTITY STATE SIMULATOR—SENDPOSITIONFRAGMENT.KT

```kotlin
package edu.nps.moves.bernd.thesis.fragments

import android.annotation.SuppressLint
import android.location.Location
import android.os.Bundle
import android.os.Looper
import android.os.StrictMode
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.Button
import android.widget.EditText
import android.widget.TextView
import android.widget.ToggleButton
import androidx.fragment.app.Fragment
import com.google.android.gms.location.*
import edu.nps.moves.bernd.thesis.R
import edu.nps.moves.dis7.*
import edu.nps.moves.dis7.enumerations.Country
import edu.nps.moves.dis7.enumerations.EntityKind
import edu.nps.moves.dis7.enumerations.ForceID
import edu.nps.moves.dis7.enumerations.PlatformDomain
import edu.nps.moves.dis7.utilities.CoordinateConversions
import java.io.ByteArrayOutputStream
import java.io.DataOutputStream
import java.net.DatagramPacket
import java.net.DatagramSocket
import java.net.InetAddress
import java.net.MulticastSocket
import java.util.*

/**
 * SendPositionFragment
 * Title:  Thesis Prototype
 * Author: LTC Bernd Weissenberger
 * Date:   03/02/2021
 *
 * Creates the SendPositionFragment of the app.
 */
class SendPositionFragment : Fragment(R.layout.fragment_send_position) {

    private val FRACTIONAL_FORMAT = "%.4f"

    // Variables for the position af mobile device
    private var latitude = 0.0
    private var longitude = 0.0
    private var altitude = 0.0

    // Text variables for output of coordinates on display
    private var latitudeValue: TextView? = null
    private var longitudeValue: TextView? = null
    private var altitudeValue: TextView? = null

    // primarily used for debug outputs on display
    private var outputValue: TextView? = null

    // to get access to the GPS data of the mobile device
    private var fusedLocationProviderClient: FusedLocationProviderClient? =
null

    override fun onStart() {
        super.onStart()
        registerForLocationUpdates()
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?,
    ): View? {
        val view: View = inflater.inflate(R.layout.fragment_send_position,
container, false)
        latitudeValue = view.findViewById(R.id.latitude_value)
        longitudeValue = view.findViewById(R.id.longitude_value)
        altitudeValue = view.findViewById(R.id.altitude_value)
        outputValue = view.findViewById(R.id.output)
        return view
    }

    override fun onStop() {
        unregisterForLocationUpdates()
        super.onStop()
    }

    fun updatePosition(location: Location) {
        latitude = location.latitude
        longitude = location.longitude
        altitude = location.altitude

        val latitudeString = createFractionString(latitude)
        val longitudeString = createFractionString(longitude)
        val altitudeString = createFractionString(altitude)
        latitudeValue!!.text = latitudeString
        longitudeValue!!.text = longitudeString
        altitudeValue!!.text = altitudeString
    }

    // build a proper string for output
    private fun createFractionString(fraction: Double): String {
        return java.lang.String.format(Locale.getDefault(), FRACTIONAL_FORMAT,
fraction)
    }

    @SuppressLint("MissingPermission")
    fun registerForLocationUpdates() {
        val locationProviderClient = getFusedLocationProviderClient()
        val locationRequest = LocationRequest.create()
        val looper = Looper.myLooper()
        locationProviderClient.requestLocationUpdates(locationRequest,
locationCallback, looper)
    }

    private fun getFusedLocationProviderClient(): FusedLocationProviderClient
{
        if (fusedLocationProviderClient == null) {
            fusedLocationProviderClient =
                LocationServices.getFusedLocationProviderClient(activity!!)
        }
        return fusedLocationProviderClient!!
    }

    fun unregisterForLocationUpdates() {
        if (fusedLocationProviderClient != null) {

fusedLocationProviderClient!!.removeLocationUpdates(locationCallback)
        }
    }

    private val locationCallback: LocationCallback = object :
LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult) {
            super.onLocationResult(locationResult)
            val lastLocation: Location = locationResult.lastLocation
            updatePosition(lastLocation)
        }
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        super.onCreate(savedInstanceState)

        val button_pzmediumblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_KPz_blue)
        val button_pzmediumred =
view.findViewById<ToggleButton>(R.id.button_NotPressed_KPz_red)
        val button_sanblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_San_blue)
        val button_sanred =
view.findViewById<ToggleButton>(R.id.button_NotPressed_San_red)
        val button_spzblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_SPz_blue)
        val button_spzred =
view.findViewById<ToggleButton>(R.id.button_NotPressed_SPz_red)

        val startButton = view.findViewById<Button>(R.id.startButton)
        val stopButton = view.findViewById<Button>(R.id.stopButton)

        button_pzmediumblue.setOnCheckedChangeListener { _, _ ->
            if(button_pzmediumblue.isChecked){
```

```
            button_pzmediumred.isChecked = false
            button_sanblue.isChecked = false
            button_sanred.isChecked = false
            button_spzblue.isChecked = false
            button_spzred.isChecked = false
        }
    }
    button_pzmediumred.setOnCheckedChangeListener { _, _ ->
        if(button_pzmediumred.isChecked) {
            button_pzmediumblue.isChecked = false
            button_sanblue.isChecked = false
            button_sanred.isChecked = false
            button_spzblue.isChecked = false
            button_spzred.isChecked = false
        }
    }
    button_sanblue.setOnCheckedChangeListener { _, _ ->
        if(button_sanblue.isChecked) {
            button_pzmediumred.isChecked = false
            button_pzmediumblue.isChecked = false
            button_sanred.isChecked = false
            button_spzblue.isChecked = false
            button_spzred.isChecked = false
        }
    }
    button_sanred.setOnCheckedChangeListener { _, _ ->
        if(button_sanred.isChecked) {
            button_pzmediumred.isChecked = false
            button_sanblue.isChecked = false
            button_pzmediumblue.isChecked = false
            button_spzblue.isChecked = false
            button_spzred.isChecked = false
        }
    }
    button_spzblue.setOnCheckedChangeListener { _, _ ->
        if(button_spzblue.isChecked) {
            button_pzmediumred.isChecked = false
            button_sanblue.isChecked = false
            button_sanred.isChecked = false
            button_pzmediumblue.isChecked = false
            button_spzred.isChecked = false
        }
    }
    button_spzred.setOnCheckedChangeListener { _, _ ->
        if(button_spzred.isChecked) {
            button_pzmediumred.isChecked = false
            button_sanblue.isChecked = false
            button_sanred.isChecked = false
            button_spzblue.isChecked = false
            button_pzmediumblue.isChecked = false
        }
    }

    // to allow sockets in main :)
```

```
        val policy = StrictMode.ThreadPolicy.Builder().permitAll().build()
        StrictMode.setThreadPolicy(policy)

        // Thread for sending PDUs in background. GUI is still working
        class BackGround : Thread(){
            var done: Boolean = false

            override fun run(){
                while (!done) {
                    sendOnePacket(view)
                    Thread.sleep(1000)
                }
            }
            fun shutdown(){
                done = true
            }
        }

        var t = BackGround()


        startButton.setOnClickListener {
            if(stopButton.isActivated)
                stopButton.isActivated = false
            t = BackGround()
            t.start()
        }

        stopButton.setOnClickListener {
            t.shutdown()
        }
    }


    private fun sendOnePacket(view: View) {
        val ipAddress = view.findViewById<EditText>(R.id.ipTextField)
        val port = view.findViewById<EditText>(R.id.portTextField)

        try {
            val socket = MulticastSocket()
            val datagramSocket = DatagramSocket()
            datagramSocket.broadcast = true
            // creating the pdu
            var espdu = EntityStatePdu()
            var espduLocation = Vector3Double()
            espdu.exerciseID = 1.toByte()
            var entityID = espdu.entityID
            entityID.siteID = 1.toShort()
            entityID.applicationID = 1.toShort()
            entityID.entityID = 1.toShort()


            // get and set location
            var entityLocation = espdu.setEntityLocation(espduLocation)
            var disCoordinates =
CoordinateConversions.getXYZfromLatLonDegrees(
                latitude,
                longitude,
                2.0
            )
            espduLocation.x = disCoordinates [0]
            espduLocation.y = disCoordinates [1]
            espduLocation.z = disCoordinates [2]
            espdu.entityLocation = espduLocation

            val button_pzmediumblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_KPz_blue)
            val button_pzmediumred =
view.findViewById<ToggleButton>(R.id.button_NotPressed_KPz_red)
            val button_sanblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_San_blue)
            val button_sanred =
view.findViewById<ToggleButton>(R.id.button_NotPressed_San_red)
            val button_spzblue =
view.findViewById<ToggleButton>(R.id.button_NotPressed_SPz_blue)
            val button_spzred =
view.findViewById<ToggleButton>(R.id.button_NotPressed_SPz_red)

            var forceID = ForceID.OPPOSING
            var entityType = EntityType()
            var marking = EntityMarking()
            if (button_pzmediumblue.isChecked) {
                forceID = ForceID.FRIENDLY
                entityType.country = Country.GERMANY_DEU
                entityType.entityKind = EntityKind.PLATFORM
                entityType.domain = Domain.inst(PlatformDomain.LAND)
                entityType.category = 1.toByte()
                entityType.subCategory = 1.toByte()
                entityType.specific = 1.toByte()
                marking.characters = “Leopard 2A6”.toByteArray()
            } else if (button_pzmediumred.isChecked) {
                forceID = ForceID.OPPOSING
                entityType.country = Country.RUSSIA_RUS
                entityType.entityKind = EntityKind.PLATFORM
                entityType.domain = Domain.inst(PlatformDomain.LAND)
                entityType.category = 1.toByte()
                entityType.subCategory = 1.toByte()
                entityType.specific = 1.toByte()
                marking.characters = “T-80”.toByteArray()
            } else if (button_sanblue.isChecked) {
                forceID = ForceID.FRIENDLY
                entityType.country = Country.UNITED_STATES_OF_AMERICA_USA
                entityType.entityKind = EntityKind.PLATFORM
                entityType.domain = Domain.inst(PlatformDomain.LAND)
                entityType.category = 2.toByte()
                entityType.subCategory = 38.toByte()
                entityType.specific = 1.toByte()
                marking.characters = “SanBoxer”.toByteArray()
```

```
        } else if (button_sanred.isChecked) {
            forceID = ForceID.OPPOSING
            entityType.country = Country.RUSSIA_RUS
            entityType.entityKind = EntityKind.PLATFORM
            entityType.domain = Domain.inst(PlatformDomain.LAND)
            entityType.category = 2.toByte()
            entityType.subCategory = 7.toByte()
            entityType.specific = 21.toByte()
            marking.characters = "MT-LB ambulance".toByteArray()
        } else if (button_spzblue.isChecked) {
            forceID = ForceID.FRIENDLY
            entityType.country = Country.GERMANY_DEU
            entityType.entityKind = EntityKind.PLATFORM
            entityType.domain = Domain.inst(PlatformDomain.LAND)
            entityType.category = 3.toByte()
            entityType.subCategory = 10.toByte()
            entityType.specific = 1.toByte()
            marking.characters = "GTK Boxer APC".toByteArray()
        } else if (button_spzred.isChecked) {
            forceID = ForceID.OPPOSING
            entityType.country = Country.RUSSIA_RUS
            entityType.entityKind = EntityKind.PLATFORM
            entityType.domain = Domain.inst(PlatformDomain.LAND)
            entityType.category = 2.toByte()
            entityType.subCategory = 1.toByte()
            entityType.specific = 1.toByte()
            marking.characters = "BMP-1".toByteArray()
        } else {
            forceID = ForceID.OPPOSING
        }

        espdu.forceId = forceID
        espdu.entityType = entityType
        espdu.marking = marking

        var timeStamp = DisTime().disAbsoluteTimestamp
        espdu.timestamp = timeStamp
        espdu.length = 144

        val baos = ByteArrayOutputStream()
        val dos = DataOutputStream(baos)

        espdu.marshal(dos)
        val data = baos.toByteArray()
        // get IP and Port from text fields
        var ina = InetAddress.getByName(ipAddress.text.toString())
        var port = Integer.parseInt(port.text.toString())

        outputValue!!.text = (espdu.length).toString()
        // create DatagramPacket
        var dp2 = DatagramPacket(data, data.size, ina, port)
        datagramSocket.send(dp2)

    } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

## D. ENTITY STATE SIMULATOR—BUILD.GRADL

```
apply plugin: ‘com.android.application’
apply plugin: ‘kotlin-android’
apply plugin: ‘kotlin-android-extensions’

android {
    compileSdkVersion 30
    buildToolsVersion “30.0.2”

    defaultConfig {
        applicationId “edu.nps.moves.bernd.thesis”
        minSdkVersion 16
        targetSdkVersion 30
        versionCode 1
        versionName “1.0”

        testInstrumentationRunner “androidx.test.runner.AndroidJUnitRunner”
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile(‘proguard-android-
optimize.txt’), ‘proguard-rules.pro’
        }
    }
    compileOptions {
        sourceCompatibility kotlin_version
        targetCompatibility kotlin_version
    }
}
android {
    compileOptions {
        sourceCompatibility 1.8
        targetCompatibility 1.8
    }
}

dependencies {
    implementation fileTree(dir: “libs,” include: [“*.jar”])
    implementation “org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version”
    implementation ‘androidx.core:core-ktx:1.3.2’
    implementation ‘androidx.appcompat:appcompat:1.2.0’
    implementation ‘androidx.constraintlayout:constraintlayout:2.0.4’
    implementation ‘com.google.android.gms:play-services-location:17.1.0’
    implementation ‘com.google.android.gms:play-services-maps:17.0.0’
    implementation ‘androidx.legacy:legacy-support-v4:1.0.0’

//    implementation ‘edu.nps.moves:open-dis7-enumerations:1.0’

    testImplementation ‘junit:junit:4.13’

    androidTestImplementation ‘androidx.test.ext:junit:1.1.2’
    androidTestImplementation ‘androidx.test.espresso:espresso-core:3.3.0’

    implementation ‘com.google.android.material:material:1.2.1’
}
```

## E. MOBILE HIT ACTOR—RASPIRECEIVER.JAVA

```
package raspi;

import edu.nps.moves.dis7.pdus.DetonationPdu;
import edu.nps.moves.dis7.pdus.EntityID;
import edu.nps.moves.dis7.pdus.Pdu;
import edu.nps.moves.dis7.utilities.PduFactory;
import java.io.File;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.MalformedURLException;
import java.net.MulticastSocket;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.sound.sampled.AudioInputStream;
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
import javax.sound.sampled.LineUnavailableException;
import javax.sound.sampled.UnsupportedAudioFileException;

/**
* Receiver for detonation PDUs
* @date 03/16/2021
* @author Bernd Weissenberger
*/
public class RaspiReceiver {

 /**
  * Main method. Will be called by the OS after
  * @param args: as usual
  */
 public static void main(String [] args) {
     // set entity ID to fix value of 321 for testing purpose.
     // in real live should be set by tip switches or changable by software.
     EntityID id = new EntityID();
     id.setEntityID(321);
     // play the detonation sound once after booting
     try {
         playSound();
     }  catch  (LineUnavailableException  |  UnsupportedAudioFileException  |
IOException | InterruptedException ex) {
         Logger.getLogger(RaspiReceiver.class.getName()).log(Level.SEVERE,
null, ex);
     }

     MulticastSocket socket;
     DatagramPacket packet;
     PduFactory pduFactory = new PduFactory();
     int pduCount = 0;
     try {
         // Specify the socket to receive data
         socket = new MulticastSocket(3000);
         while (true) // Loop infinitely, receiving datagrams
         {
             byte buffer [] = new byte [8192];
             packet = new DatagramPacket(buffer, buffer.length);
             socket.receive(packet);
             List<Pdu> pduBundle =
pduFactory.getPdusFromBundle(packet.getData(), packet.getLength());
             if (pduBundle.size() > 1) {
                 System.out.println("Bundle size is " + pduBundle.size());
             }
             // end iterator loop through PDU bundle
             for (Pdu aPdu : pduBundle) {
                 pduCount++;
                 // only handle DetonationPDU at this point
                 if (aPdu instanceof DetonationPdu) {
                     // it is a detonation PDU....
                     DetonationPdu tPDU = (DetonationPdu) aPdu;
                     if(tPDU.getTargetEntityID().getEntityID()
                          == id.getEntityID())
                         //... and it was for me, so: play detonation sound
                         playSound();
                 } else {
                     // not a detonation PDU
                     System.out.println("wrong PDU type received");
                 }
             } // end of bundle loop            } // end of while loop
     } // end try block
     catch (IOException ioe) {
         System.out.println(ioe);
     }  catch  (LineUnavailableException  |  UnsupportedAudioFileException  |
InterruptedException ex) {
         Logger.getLogger(RaspiReceiver.class.getName()).log(Level.SEVERE,
null, ex);
     }
 }

 /**
  * Method is called, if a detonation PDU for this device was received.
  */
 public     static     void     playSound()     throws     MalformedURLException,
LineUnavailableException,      UnsupportedAudioFileException,      IOException,
InterruptedException {
     File url = new File("sounds/explosion.wav");
     Clip clip = AudioSystem.getClip();

     AudioInputStream ais = AudioSystem.getAudioInputStream(url);
     clip.open(ais);
     clip.start();
     while (!clip.isRunning()) {
         Thread.sleep(3);
     }
     while (clip.isRunning()) {
         Thread.sleep(3);
     }
     clip.close();
     System.out.println("Sound played...");
 }
}
```

## F. MOBILE HIT ACTOR TESTER—SENDDETONATIONFRAGMENT.KT

```
package edu.nps.moves.bernd.thesis.fragments

import android.os.Bundle
import android.os.StrictMode
import android.view.View
import android.widget.EditText
import android.widget.ImageButton
import androidx.fragment.app.Fragment
import edu.nps.moves.bernd.thesis.R
import edu.nps.moves.dis7.DetonationPdu
import edu.nps.moves.dis7.DisTime
import edu.nps.moves.dis7.EntityID
import java.io.ByteArrayOutputStream
import java.io.DataOutputStream
import java.net.DatagramPacket
import java.net.InetAddress
import java.net.MulticastSocket

class SendDetonationFragment : Fragment(R.layout.fragment_send_detonation) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        super.onCreate(savedInstanceState)

        val detButton = view.findViewById<ImageButton>(R.id.detButton)


        // to allow sockets in main :)
        val policy =
            StrictMode.ThreadPolicy.Builder().permitAll().build()
        StrictMode.setThreadPolicy(policy)

        //detonation
        detButton.setOnClickListener {
            val ipAddress = view.findViewById<EditText>(R.id.ipTextField)
            val port = view.findViewById<EditText>(R.id.portTextField)
            try {
                val socket = MulticastSocket()
                // creating the pdu
                var detPDU = DetonationPdu()
                var timeStamp = DisTime().disAbsoluteTimestamp
                detPDU.timestamp = timeStamp
                // setting the target EntityID

detPDU.setTargetEntityID(EntityID().setEntityID(321.toShort()))

                val baos = ByteArrayOutputStream()
                val dos = DataOutputStream(baos)

                detPDU.marshal(dos)
                val data = baos.toByteArray()
                //ipAddress.toString()
                var ina = InetAddress.getByName(ipAddress.text.toString())

                var dp = DatagramPacket(data, data.size)
                dp.setAddress(ina)
                dp.setPort(Integer.parseInt(port.text.toString()))

                socket.send(dp)
                //client.close()
                socket.close()

            } catch (e: Exception) {
                e.printStackTrace()
            }
        }//end Detonation
    }

}
```

## G. MOBILE HIT ACTOR TESTER—FRAGMENT_SEND_DETONATION.XML

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".fragments.SendDetonationFragment">


    <GridLayout
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:orientation="horizontal">
        <EditText
            android:id="@+id/ipTextField"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="0"
            android:layout_column="2"
            android:ems="10"
            android:inputType="textPersonName"
            android:text="192.168.188.103" />

        <EditText
            android:id="@+id/portTextField"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_row="1"
            android:layout_column="2"
            android:ems="10"
            android:inputType="number"
            android:text="3000" />

        <ImageButton
            android:id="@+id/detButton"
            android:layout_width="160dp"
            android:layout_height="159dp"
            android:layout_row="2"
            android:layout_column="2"

            android:layout_gravity="center"
            app:srcCompat="@drawable/fire_smexplosion" />


    </GridLayout>
</FrameLayout>
```

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF REFERENCES

*Android Team Awareness Kit*. (2021). Wikipedia. https://en.wikipedia.org/wiki/Android_Team_Awareness_Kit

Arnold, K., Gosling, J., & Holmes, D. (2005). *THE Java$^{TM}$ Programming Language* (4th ed.). Addison Wesley Professional.

Barksdale, T. (2014). CAS-KILSWITCH and the way ahead. *Marine Corps Gazette*, 34–37.

Bhagwat, P. (2001). Bluetooth: Technology for short-range wireless apps. *IEEE Internet Computing*, *5*(3), 96–103. https://doi.org/10.1109/4236.935183

Bhoyar, R. P., Ghonge, M. M., & Gupta, S. G. (2013). Comparative Study on IEEE Standard of Wireless LAN/ Wi-Fi 802.11 a/b/g/n. *International Journal of Advanced Research in Electronics and Communication Engineering*, *2*(7), 5.

Bose, S. (2018). A comparative study: Java vs Kotlin programming in Android application development. *International Journal of Advanced Research in Computer Science*, *9*(3), 41–45. https://doi.org/10.26483/ijarcs.v9i3.5978

Carpenter, D., & Carpenter, A. D. (2013). *An Approach to Command and Control Using Emerging Technologies*. Air Force Research Laboratory. https://apps.dtic.mil/dtic/tr/fulltext/u2/a587481.pdf

Department of Defense. (2008). *Department of Defense Interface Standard—MIL-STD-2525C*. https://www.jcs.mil/Portals/36/Documents/Doctrine/Other_Pubs/ms_2525d.pdf

Ferro, E., & Potorti, F. (2005). Bluetooth and wi-fi wireless protocols: A survey and a comparison. *IEEE Wireless Communications*, *12*(1), 12–26. https://doi.org/10.1109/MWC.2005.1404569

Griffiths, D., & Griffiths, D. (2019). *Head first Kotlin: A brain-friendly guide*. O'Reilly Media, Inc.

Hossein Motlagh, N. (2012). *Near Field Communication (NFC)—A technical Overview* [Master's thesis, University of Vaasa]. https://doi.org/10.13140/RG.2.1.1232.0720

IEEE. (2012). *1278.1-2012—IEEE standard for Distributed Interactive Simulation—Application protocols*. IEEE. https://standards.ieee.org/standard/1278_1-2012.html

IEEE. (2015). *1278.2-2015—IEEE standard for Distributed Interactive Simulation (DIS)—Communication services and profiles*. IEEE. https://standards.ieee.org/standard/1278_2-2015.html

Informationszentrum Mobilfunk. (2020). *Daten und Fakten zu 5G*. https://www.informationszentrum-mobilfunk.de/mediathek/broschueren/daten-und-fakten-zu-5g

McGregor, D. (2011a). *Distributed Interactive Simulation (DIS)* [Presentation].

McGregor, D. (2011b). *High Level Architecture (HLA)* [Presentation].

Noohani, M. Z., & Magsi, K. U. (2020). *A review of 5G technology: Architecture, security and wide applications*. *07*(05), 34.

Nordrum, A., Clark, K., & IEEE. (2017, January 27). *Everything You Need to Know About 5G - IEEE Spectrum*. IEEE Spectrum: Technology, Engineering, and Science News. https://spectrum.ieee.org/video/telecom/wireless/everything-you-need-to-know-about-5g

Panasonic. (2020). *5 Things Military Leaders Need to Know About ATAK*. https://federalnewsnetwork.com/wp-content/uploads/2020/06/Panasonic-ATAK-Top5-WhitePaper-Final-040620-1.pdf

Sadler, L. C., & Metu, S. (2017). *Intelligent Command and Control Demonstration Setup and Presentation Instructions*. US Army Research Laboratory. https://apps.dtic.mil/sti/pdfs/AD1043277.pdf

Straßburger, S. (2006). Overview about the High Level Architecture for modelling and simulation and recent developments. *Simulation News Europe*, *16*, 5–14.

Tailor, N. (2015). *Brief about USB 3.0 and Comparison with USB 2.0*. https://doi.org/10.13140/RG.2.1.4095.8885

*TIOBE*. (n.d.). Retrieved February 14, 2021, from https://www.tiobe.com/tiobe-index//

*USB*. (n.d.). Retrieved May 5, 2021, from https://en.wikipedia.org/wiki/USB

Watson, R. (2012). *Understanding the IEEE 802.11ac Wi-Fi Standard*. MERU Networks. https://eddywireless.com/yahoo_site_admin/assets/docs/2012-wp-ieee-802-11ac-understanding-enterprise-wlan-challenges.9141800.pdf

Woolley, M. (2019). *Bluetooth Core Specification v5.1*. Bluetooth. https://www.bluetooth.com/wp-content/uploads/2019/03/1901_Feature_Overview_Brief_FINAL.pdf

Ylinen, J., Koskela, M., Iso-Anttila, L., & Loula, P. (2009). Near Field Communication Network Services. *2009 Third International Conference on Digital Society*, 89–93. https://doi.org/10.1109/ICDS.2009.43

You, Y., Lee, T., Kim, W., & Yoon, S. (2016). Development of an OMT Table Viewer/ Editor Using the Matlab/Simulink for HLA-Based Distributed Simulation. *International Journal of Information and Electronics Engineering*, *6*(2), 89–92. https://doi.org/10.18178/IJIEE.2016.6.2.601

THIS PAGE INTENTIONALLY LEFT BLANK

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California