Preface

This thesis is written in order to complete our master education in electronics and ICT engineering at UHasselt Belgium. The development and writing of this thesis has been done in the Tampere University of Technology, Finland. K. Aerts was the supervisor of our home university, Prof. J. Nurmi and PhD student K. Wang supervised us during the exchange period.

After several meetings with our supervisors we had a solid idea of what the thesis should be about and what the research question should be. We had some major problems with multiple software versions, but thanks to dedicated research and trial and error we were able to fix all of these issues. We would like to thank our supervisors to contribute in our work and keep us motivated along the whole experience.

Thanks, to all our family members who visited us here in Finland to provide us with some comfort after surviving the cold winter and to all new friends we met here. Without the other exchange students, this experience would not have been this great.

Lander Beckers
Henning Lakiere

Tampere, 02/06/2017

Chapter 1

Theoretical background

1.1 parallel computing

1.1.1 What is parallel computing

The problem when facing large calculations is that they require a lot of computing power and thus time. Performing these calculations can be done with two types of computation, serial and parallel computing. Serial computing means, you have one compute unit (e.g. a single core CPU) available that performs all instructions on a certain set of data. This set of data will be broken into multiple smaller subparts that will be solved by a certain instruction. The single compute unit performs the instruction on every subpart in order to solve the whole part as shown in figure 1.1 [1].

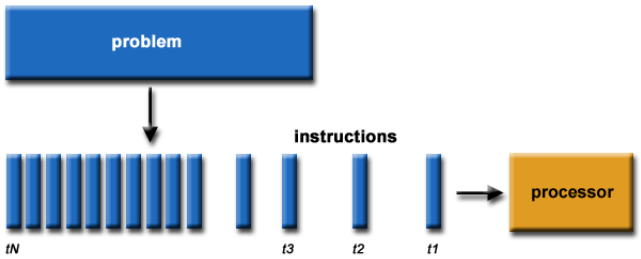


Figure 1.1: Workflow serial computing

Parallel computing, on the other hand, is the simultaneous use of multiple compute units, or a compute resource, to solve a computational problem. This compute resource can be a CPU with multiple cores, a combination of a CPU with different compute accelerators such as a GPU or even a whole network of computers and servers. We break apart the main problem in smaller subproblems as we did with serial computing. Now, since there are multiple compute units, we can distribute the subproblems among all these compute units. Every unit can now perform an instruction on their given

subproblem simultaneously as shown in figure 1.2. With multiple compute

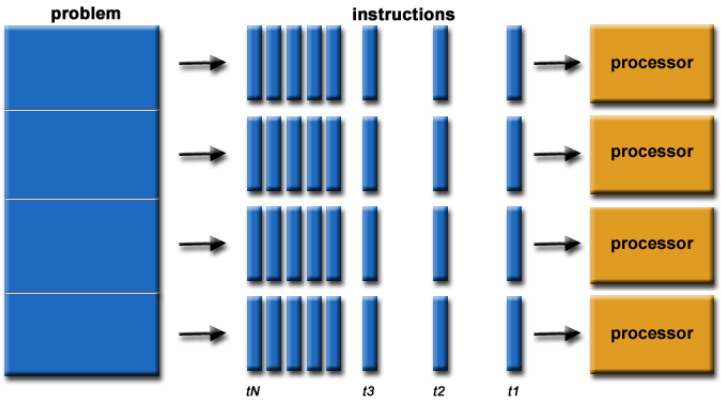


Figure 1.2: Workflow parallel computing

units executing one task, we will shorten the completion time and even have a potential cost saving. It also allows us to solve larger/complex problems since a single computer could suffer from limited memory. And last, we are able to access non-local resources in a network that would not be accessible from a local computer.

It is easy to conclude that the concept of parallel computing was to have a more efficient way to handle large sets of data such as huge databases, images or simulations that involve large datasets. It is also easier to deal with complex data for example algorithms [1].

1.1.2 Classification of parallel computing

Parallel computing systems can be separated into different classes. According to Flynn's taxonomy, we can roughly place any of these systems in one of the four classes. This classification was first studied by Michael Flynn in 1972 [2]. The classifications are determined by two factors: instruction stream and data stream which both have two possible states being single or multiple. Figure 1.3 represents a the four possible classes from the Flynn's taxonomy [1].

- SISD

  The SISD class will have a single core processor executing a single data stream to operate on data stored in a single memory (figure 1.4a). This means that a parallel compute system cannot be classified as an SISD system, but Flynn's taxonomy was not made for just classifying parallel compute systems. Any traditional single-core processor falls into this category but it is usually old computers or older compute units that can be classified as SISD.

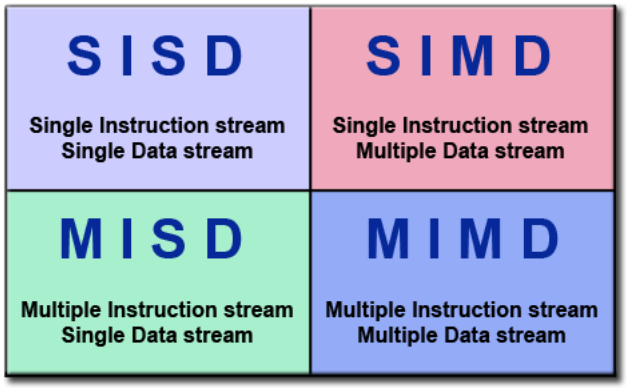


Figure 1.3: Four possible classifications according to Flynn's Taxonomy

- SIMD

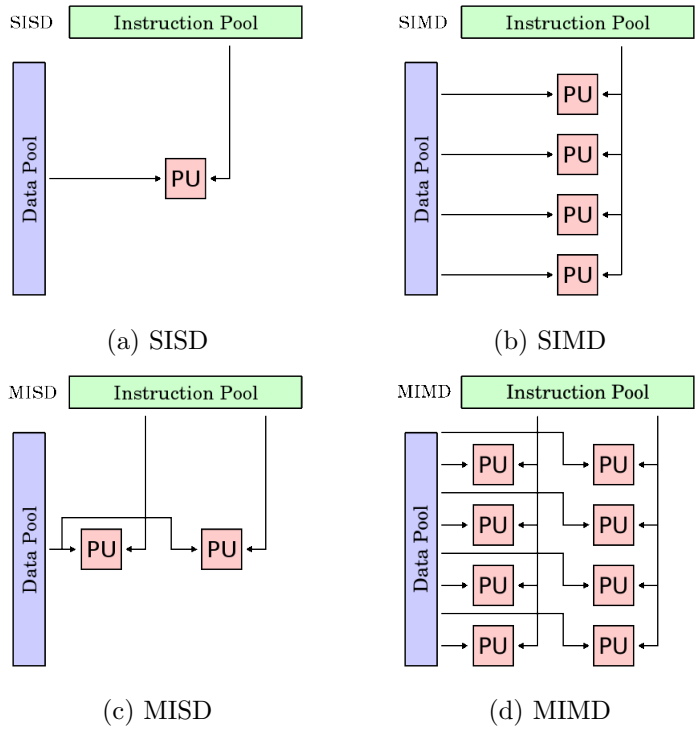
  Data is distributed amongst multiple processors who all execute the same instruction on this data (figure 1.4b). Since we have access to multiple compute units, parallel computing can be categorized in this class. Furthermore, this is the class where we can categorize our subject of the thesis in since we have a large set of data divided over multiple data streams (MD) and only one instruction stream (SI) since all processing units will perform the same instruction on the data. The SIMD class contains the most modern computers, particularly those with a graphics processor unit (GPU).

- MISD

  Each processing unit operates on the data independently via separate instruction streams while a single data stream is fed into multiple processing untis (figure 1.4c). This class knows very few applications. An example of an application is the use of multiple cryptography algorithms attempting to crack a single coded message.

- MIMD

  This time every processing unit is able to execute a different instruction stream on a different data stream (figure 1.4d). This means that any instruction can be applied on any data package for every compute unit. Most supercomputers, networked parallel computer clusters and "grids" can be classified as a MIMD compute system. Also, many MIMD architectures include SIMD execution sub-components.



(a) SISD (b) SIMD

(c) MISD (d) MIMD

Figure 1.4: Architecture classes from Flynn's taxonomy

1.1.3 OpenCL

Companies worldwide constantly strive to improve computational performance. They start using GPUs, FPGAs and other compute accelerators that behave as a coprocessor to process parallel workload. In order for these heterogeneous architectures to function properly, we need software that supports heterogeneous computing on hardware platforms from different vendors. To make this possible, developers use toolkits such as Threading Building Blocks (TBB), OpenMP, Compute Unified Device Architecture (CUDA), and others [3]. However, some of the existing toolkits were limited to either only being able to use a single microprocessor family or they did not support heterogeneous computing. OpenCL on the other hand provides a set of easy-to-use abstractions and a wide variety of APIs. OpenCL was developed by the Khronos group as a parallel computing API for Apple's OS X release of Snow Leopard back in 2009. This Khronos group is a mixture of people from different hardware vendors like ATI technologies, Intel, Nvidia to name a few [4].
One of the main reasons companies start using OpenCL is that in the past they would use GPUs, when they should be using FPGAs while others had the problem the other way around. The problem they had is that converting

CUDA, which is the parallel computing platform from Nvidia (GPUs), to VHDL is difficult and annoying to do. More reasons on why OpenCL should be used to program FPGAs are listed below [**?**].

- Simplicity and ease of development
  Because most software developers are more familiar with C than low-level HDL language, OpenCL is easy to understand for the vast majority of developers worldwide.

- Code profiling
  OpenCL allows you to determine where exactly the performance-sensitive pieces in your code are. This way it is easy to assign these pieces of code to be executed by hardware accelerators as kernels.

- Performance & Efficiency
  Every developer wants to have his software build in the most efficient way to benefit from maximum performance. Due to the FPGA's parallelistic architecture, you only need to generate the logic the device needs to run the software to deliver high performance.

- Code reuse
  Since there are multiple devices that are supported by OpenCL, you can reuse your already written code on almost any of the other devices without having to change a thing.

1.2 Smartphone

Nowadays, almost everyone has a smartphone. These portable computers allow us to communicate with anyone across the world from almost any place. They can be used as entertainment devices to play music or video games and over the last couple of years even services like trading markets or banking systems have been integrating with smartphones. Furthermore, you can make pictures with them, view all sorts of media, and the list goes on. Since the increasing popularity there has been many companies developing smartphones. But the most common brands are Samsung and Apple accounting for over 37% of the market shares in 2017 of all smartphone brands [5].

1.2.1 Android OS

Like most communication devices, smartphones need an operating system (OS). While Apple's IPhones use their own OS called iOS, Samsung and many other smartphone manufacturers use the Android OS that was developed by Google. With a whopping 81,7% worth of market shares at the end of 2016, Android is definitely one of the market leaders when it comes to

smartphone operating systems [6].
One of the main features of smartphones are applications or apps short. Google's Play Store has over a million apps available for almost any Android device. Unlike Apple's App Store, basically anyone can upload their own apps on Google's Play Store. To upload apps as a developer for everyone to download and use you pay a one-time fee of 25$ to Google. Making apps on the other hand is free. Most apps are written in Java and there are multiple integrated development environments (IDE) available for the Android platform. The official IDE for Android is Android Studio which is described in the next section. Other IDEs available are AIDE (HTML, C, C++), Xamarin (C#) and many others.

1.2.2 Android Studio

Android Studio is the official IDE where you can develop apps for phones with an Android OS. Its main programming language is Java but since Android Studio 2.2 it is possible to write and use C and C++ code by compiling it into a native library. With the Java Native Interface (JNI) you can call the C/C++ functions in your native library. Furthermore, Android Studio splits up front and back end of the application giving the developer a nice clean overview of the whole project. The front end can be edited through coding or with a visual interface where you can pick and place your required objects in a layout. The front end design is an xml-file that is attached to an activity. This activity is part of the back end where you write your code in order to interact with the front end of the application.

1.3 Bluetooth

Bluetooth is a form of wireless communication that was developed in 1994 by Ericsson Mobile in Sweden. It is a radio frequency (RF) technology using the 2.4GHz industrial, scientific and medical (ISM) band, the same band where you can find ZigBee and WiFi aswell. It can be used to transmit data or voice communication over short distances. Bluetooth radios can be found in nearly every new smartphone and laptop device. It is easy to use, to setup and it has a lot of applications, for example hands-free devices, home heating systems, entertaining devices and so on. Bluetooth is designed to be low cost, for about 5-10$ per unit. The down side of this is the short connection range and the limited transmission speed of around 780 kb/s [7].

1.3.1 Bluetooth benefits

The introduction of Bluetooth allowed for many new applications in several areas. Even today it is still widely used, mostly for multimedia devices,

keyboards, mices, printers. The following list explains some benefits for three general areas:

Data and voice acces points. Bluetooth allows a wireless connection between devices through which they can communicate. With Bluetooth, the devices are able to transmit voice and data packages in real-time.

Cable replacement. Some wired connections between devices require special cables or adapters. Bluetooth eliminates this hassle since any device can connect to another with the right communication protocol. The range of this connection is approximately 10m and doesn't require the devices to be in line of sight. With an optional amplifier the range can be extended to 100m.

Ad hoc networking. Devices with a Bluetooth radio can establish instant connections with each other as soon they come into range.

1.3.2 Master, slave and piconet

For a Bluetooth connection to exist, there has to be at least one master and one slave device. They use what is called the master/slave model. A master device can be connected to up to seven slave devices while a slave can only connect to one master device. A network of one master and one to seven slaves is called a piconet. The master device will coordinate all the communication throughout the piconet. All slave devices are allowed to exchange data with the master device when granted premission, but cannot communicate with other slaves in the piconet. The connection between each device is encoded and protected to prevent other devices from eavesdropping and to prevent interference between other devices. Furthermore, in order for these devices to connect with each other, they require the same communication protocol. A device in one piconet can also exist as part of another piconet and can function as either a slave or a master in each piconet. This form of overlapping is called a scatternet [8]. An example of two piconets forming a scatternet is shown in figure 1.5.

The piconet/scatternet allows the devices to share the same physical area, allowing the network to make efficient use of the bandwidth. A Bluetooth system can use up to 79 different frequencies using a frequency hopping (from 2.402 to 2.480 GHz) [9] scheme with a carrier spacing of 1MHz. This allows a bandwidth of 80MHz. Without frequency hopping scheme, every single channel would have a bandwidth of 1MHz at their disposal. With frequency hopping, the sequence will define a logical channel. This allows to have an available bandwidth of 1MHz at any given time, with a maximum of eight devices sharing the bandwidth. This 80 MHz bandwidth can be shared by several different logical channels. Though, this can cause signal

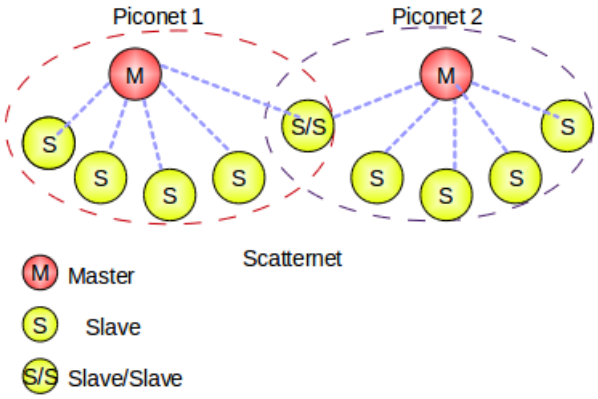


Figure 1.5: Piconets and Scatternets

collisions when devices in different piconets, on different logical channels have the same hop frequency at a given time. Signal collisions degrade the performance, so we can state that the more piconets we have, the more collisions occur, the lower our total performance will be [8].

1.3.3 Protocol architecture

The Bluetooth protocol architecture consists of four basic layers: core protocols, cable replacement, telephony control protocols and adopted protocols. Figure 1.6 shows the architecture of the Bluetooth protocol stack.

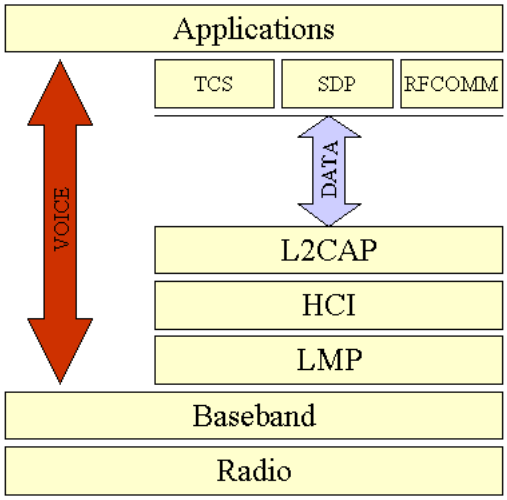


Figure 1.6: Bluetooth protocol stack

Core protocols. The core protocol is a five-layer stack. Every layer in the stack has its own responsibilities that are mentioned below.

- The *radio* layer is the wireless connection that specifies certain details about the air interface, including frequency, the use of frequency hopping, modulation scheme and transmit power.
- The *baseband* layer is responsible for the packet transmission to the radio layer. As mentioned before this data can contain data or voice packages. For the data packages, asynchronous connectionless (ACL) links are used while voice packages are transmitted with synchronous connection-oriented (SCO) links. The baseband layer maintains both ACL and SCO links. It is important for data packages to be transmitted correctly to maintain data integrity, while it is not a problem in case some voice packages get lost. That is why SCO packages are never retransmitted. If you would retransmit voice packages, every next package would suffer from a time delay restraining us from having real-time communication.
- The *Link manager protocol (LMP)* uses the links setup by the baseband and manages the connection between Bluetooth devices. Furthermore, it is responsible for monitoring service quality, security aspects such as device authentication, encryption plus the control and negotiation of baseband packet sizes.
- The *Host controller interface (HCI)* is the layer between the hardware and the software. The L2CAP layer and the layers above it are implemented in the software while all other layers under the HCI (LMP, baseband, radio) are part of the hardware. The HCI driver acts as a physical bus that connects the hardware with software. It is possible to access the L2CAP layer directly by the application making it easier for application programmers. This makes the HCI, in some cases, an unnecessary component.
- The *Logical link control and adaptation protocol (L2CAP)* receives application data and transforms this to the Bluetooth format. Furthermore, Quality of Service (QoS) parameters are exchanged at this layer [9].
- According to [9], the *Service discovery protocol (SDP)* is not a part of the Core protocols. Though, it contains all the information, services and charasteristics in order to establish a connection between two or more Bluetooth devices. The LMP uses the SDP's first to find out what services are available from the acces point. Then information from the SDP is obtained by the LMP to create a L2CAP channel.

Cable replacement. The RFCOMM seen in figure 1.6 is the cable replacement protocol. It is a virtual serial port that is designed to replace cable technology. Serial ports are common types of communication interfaces used with computing and communication devices [8]. So with RFCOMM we eliminate the need for serial ports for communication between two devices, assuming both are equiped with a Bluetooth radio. EIA-232, once known as RS-232, is a widely used serial port interface standard. The RFCOMM will provide binary data transport and has to emulate EIA-232 control signal to the baseband layer.

Telephony control protocols. Telephony control specifications (binary) or TCS BIN, is a bit-oriented protocol that is necessary to define the call control services in order to establish speech or data calls between the Bluetooth devices.

Adopted protocols. Adopted protocols are protocols developed by other organizations. They are "adopted" into the overall Bluetooth architecture. They are usually standard protocols well known in applications other than Bluetooth. Bluetooth's strategy is to only invent necessary protocols and use existing standard protocols whenever possible. The following standards are the adopted protocols:

- The *PPP*, or point-to-point protocol is as a internet standard protocol for transporting IP datagrams over point-to-point links.
- *TCP/UDP/IP*, are the foundation protocols of the TCP/IP protocol suite.
- *The object exchange protocol*, or OBEX, is a session level protocol made by the Infrared Data Association (IrDA). It is used for exchaning objects. OBEX comes with quite similar funcionalities as HTTP, but in a simpler way. There is also a model included in OBEX that is used for the representation of objects and operations.
- Bluetooth also adopts the wireless application environment *WAE* and the wireless application protocol *WAP* into its architecture.

1.4 Web sockets

Web sockets are used for fast, real-time communication between a server and a client. The HTTP model, which is also a communication protocol between a server and a client, allowed the client only to request data from the server, while the server was only able to fulfill these requests. Web sockets on the other hand, allow bidirectional communication between the

server and the client. This means both client and server can request data and also respond to these requests. The main point of web sockets is to have true concurrency and to focus on the optimization of performance when it comes to communication and exchanging data. The web socket protocol that will be discussed is also known as the RFC6455 model [10].

1.4.1 WebSocket communication

Handshake. To communicate over web socket, the server and the client first have to connect with each other. The establishment of a web socket connection is done by a web socket handshake. Handshaking is the exchange of information between two devices and the agreement about which protocol will be used to exchange data after the connection is established. A well known example of this is the TCP three-way-handshake. The client sends a synchronization message (SYN) to the server as a request to synchronize with the server. If the server receives this message and allows the client to synchronize with it, it will send a similar SYN message and an acknowledgement message (ACK) to let the client know that it has received the request. When the client receives the SYN-ACK message, it will send an ACK message back to the server to acknowledge that it has received the server's message. The TCP connection is established whenever the server receives the ACK message from the client.
The web socket handshake is quite similar. The client sends a handshake request to the server, and the server will respond with a similar handshake request as seen in figure 1.7. The desktop, smartphone and tablet in figure 1.7 represent the clients that are connected to the server. The handshake request from the client-side is shown in listing 1.1, while the response from the server is shown in listing 1.2.

```
1| GET /chat HTTP/1.1
2| Host: example.com:8000
3| Upgrade: websocket
4| Connection: Upgrade
5| Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6| Sec-WebSocket-Version: 13
```

Listing 1.1: Client's request for WebSocket handshake

Listing 1.1, the handshake request from the client, is a pretty standard HTTP request. It is built with multiple headers, some of which are mandatory for the request to be valid. If one of the headers is not understood, the server will reply with "400 Bad Request" and it will close the socket afterwards. In some cases, it will also give a reason why the handshake failed, although browsers do not display these messages. If there is a problem

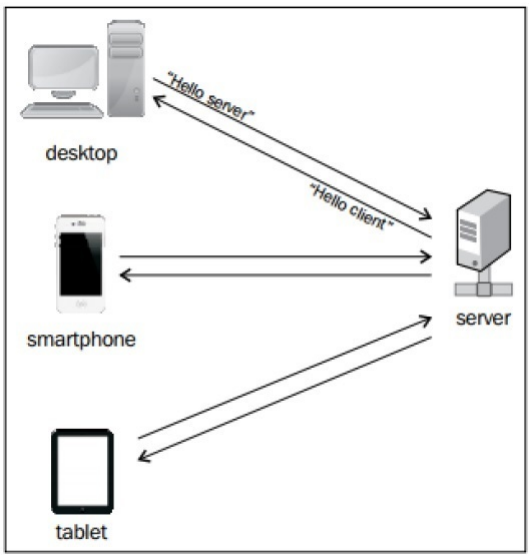


Figure 1.7: Web socket handshaking

with version numbers, the server adds a "Sec-WebSocket-Version" header in the HTTP response that contains the version it understands [11]. When

```
1| HTTP/1.1 101 Switching Protocols
2| Upgrade: websocket
3| Connection: Upgrade
4| Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5|
```

Listing 1.2: Server's response for WebSocket handshake

the server receives a request handshake from a client with all the necessary headers, it will reply with a HTTP response as shown in listing 1.2. The "Sec-WebSocket-Accept" header is derived from the "Sec-WebSocket-Key" header from the client's handshake request. To get it, we combine the "Sec-WebSocket-Key" header and "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" together. The second string is "a magic string". A magic string is predefined by the developer. It is made in a way where you would not expect it to be received from an input. When the "Sec-WebSocket-Key" header and the magic string are combined, the SHA-1 hash is taken from the result, and the base64 encoding of the hash is returned [11]. The SHA-1 is a cryptographic hash function, while base64 is a binary-to-text encoding scheme.

WebSocket URI's WebSocket defines two URI schemes. You can either use ws or the wss scheme. The ws (web socket) is a regular connection sim-

ilar to http. While wss (web socket secure) is a secured connection similar to https. The schemes are built as follows:

ws-URI = "ws:" "//" host [":" port] path ["?" query]
wss-URI = "wss:" "//" host [":" port] path ["?" query]

The most important components of the ws or wss are the host and its port. The host is determined by the server's IP address, while the port defines which port the server uses for the communication. If there is no specified port, the standard port used for ws is 80, and the standard port used for wss is 443.

Data Frames. The main advantage of WebSockets is bidirectional communication. So at any point in time, either the client or the server can send a message. Every data frame that is sent from the client to the server, or vice versa, follows the same format as seen in figure 1.8.

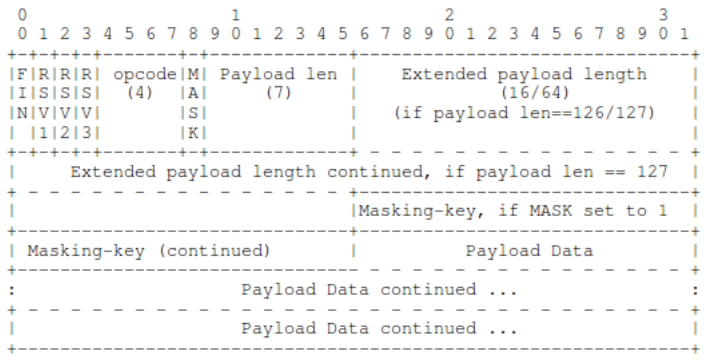


Figure 1.8: Standard WebSocket dataframe format

- FIN: 1 bit

Depending on the value of this bit, it either tells the receiving end whether or not this is the final fragment of the message. If the bit equals "0", it is not the last fragment and the receiver will continue listening for more fragments. If the bit equals "1", it means it is the last fragment of the message and the server will consider the message being delivered.

- RSV1, RSV2, RSV3: 1 bit each

All of these bits are reserved for WebSocket extensions. They should be "0" unless the client and server negotiated on whether or not a specific extension requires the use of any of the three bits. If any of these three bits is not zero while the client did not negotiate on any of these bits being non-zero, the receiving end will "fail" the WebSocket connection.

- Opcode: 4 bits

These 4 bits will define how the receiving end should interprete the data. If the receiving end does not understand the opcode it will, as in the previous case, "fail" the WebSocket connection. The information about the different opcodes is found at [12].

x0: continuation frame; this frame contains data that should be appended to the previous frame

x2: binary frame; this frame (and any following) contains binary data

x3 - x7: non-control reserved frames; these are reserved for possible WebSocket extensions

x8: close frame; this frame should end the connection

x9: ping frame

xA: pong frame

xB - xF: control reserved frames

- Mask bit: 1 bit

This bit tells wether or not the frame uses a mask. If this bit is set to "1", a masking key is included in the message. This masking key is used to unmask the data in the payload. Every frame that is sent from the client to the server must have this bit set to "1".

- Payload length: 7 bits, 7+16 bits, 7+64bits

The seven bits determine the length of the payload. If these seven bits equal 126, or "1111110", the actual length is determined by bits 16 to 31 (so 16 extra bits). These are the following 2 bytes. If the seven bits equal 127, or "1111111", the actual length is determined by bits 16 to 79 (so 64 extra bits). These are the following 8 bytes.

- Masking key: 4 bytes

As mentioned previously, this field only exists if the mask bit is set to one. All the messages who have this field set to one, are masked by a 32-bit value. If the mask bit is set to zero, there will be no masking key in the first place.

- Payload data: x+y bytes

The payload data is the combination of the extension data and the application data. These two are listed below.

- Extension data: x bytes

The extension data is non-existent unless it was negotiated on the opening handshake between the server and the client. As mentioned earlier, the RSV1-3 bits are responsible for these extensions. Any extension that has been negotiated by the client and server must have a specified length of the "Extension data". It can also tell the receiving end on how to calculate this length. As said previously, the extension is part of the total payload data.

- Application data: y bytes

The application data contains the actual data that has be to transmitted. It takes up the remaining space in the frame after any extension data. The application data is, like the extension data, part of the payload data.

Chapter 2

Implementation

2.1 Communication protocol

If we want to communicate between the devices, we have to define a set of rules that each of the devices have to follow. These rules should be made in a way that 100% of the transmitted data will arrive at the receiving end. For example, if data gets lost through transmission, the device on the receiving end will not receive all the necessary data. This can be prevented by adding a checksum that is different then from all other receivable data. If the received message looks incorrect, the receiving device has to respond by asking to resend this piece of data. We are trying out two different communication technologies so we have to set up a communication protocol for each one of them since they both have different performance aspects. To clarify future images, matrix A and B are the matrices sent to the Soc in order to calculate their multiplication. Matrix C is the calculated matrix send back to the phone.

2.1.1 Bluetooth communication protocol

To define a set of rules we first analyzed how good the Bluetooth transmission is. We did some testing and came to the conclusion that there was quite a lot of data loss. We first tried sending eight characters at once but nearly 80% of the time there was atleast one of the characters missing. Missing one number would mean that the matrix we need is incorrect and this would lead to an incorrect outcome. We did some further testing and realized it was the best option to send four numbers at a time. With this we can set up our first rule. The first rule being, we let both devices know that they should always send four numbers at once. Both devices also know they should receive four numbers. If this is not the case we have to resend the message. In order to make sure all characters would arrive at the receiving end, we added an extra character ':' at the end of each message. This character lets the receiving end know that it is the end of a message. If we would not

define the end of a message and a number would be missing, the receiving end would wait for the next number to come through. The problem is that the next number would be part of the next message causing the all numbers to be messed up with each other. An example of a message would look like this: 4372: .
Now for the communication, we start off by connecting both devices with Bluetooth. As soon as the devices are connected with each other, the SoC will send message "C" to the Android phone notifying it is ready to receive the two matrices. As soon as the phone receives message "C", it will start sending the first number or in some cases a part of the number. In case the message is received correctly, the SoC sends the message "O" to let the phone know the message was received. Figure 2.1 shows an example of how a correct message should look like.

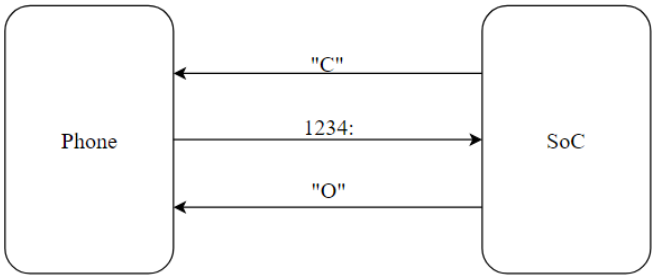


Figure 2.1: Correct message transfer

In case we are missing a number due to data loss, the receiving end will respond with the message "R" as shown in figure 2.2

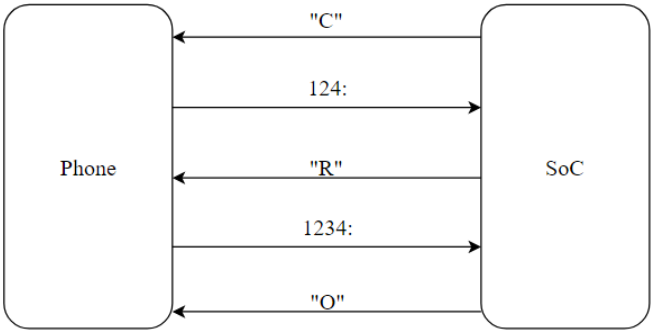


Figure 2.2: Incorrect message transfer

There is also the possibility of a device not responding to a request. The receiving end is waiting for a response from its request. If the wait time is too long compared to a predefined waiting time, the receiving end will send a message "R" as seen in figure 2.3.

As for now, we work with square matrices with a predefined size. Both

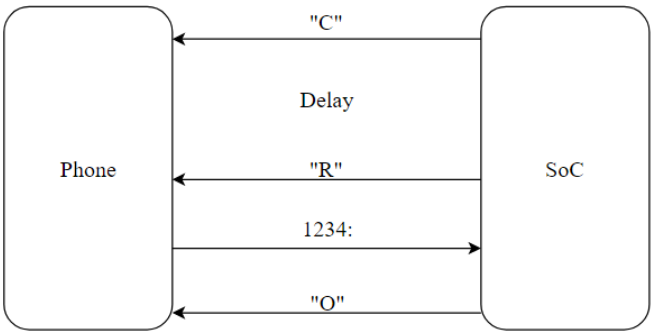


Figure 2.3: Delayed message

devices know at the start of the session how large both matrix A and matrix B are. This way the SoC can calculate how many numbers it should receive from the phone. Whenever matrix A is transferred succesfully, the Soc will send message ”D” to notify the phone it should start sending matrix B (Figure 2.4). The same procedures as mentioned above are applied on matrix B as well.

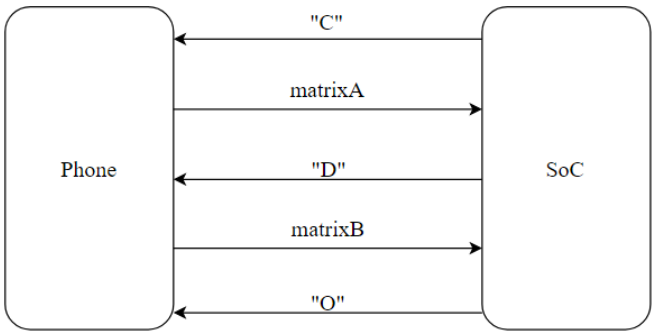


Figure 2.4: Switch matrix message

When both matrices are transferred, the role of transmitter and receiver are switched. The phone will now act as a receiver while the SoC becomes the transmitter. The SoC will transmit matrix C after doing the multiplication of matrices A and B. For this proces, the previous rules are also applicable. Although this time, the phone will start off by sending a message ”C” to notify the SoC that it is ready to receive matrix C. The SoC proceeds by sending the data in the same way as the phone did. Whenever the transmission experiences data loss, the phone will send message ”R” to the SoC exactly like the SoC does the other way around. Message ”D” is not used by the phone since there is only one matrix it should receive from the SoC. When the phone received matrix C, the Bluetooth connection between the devices will be suspended.

2.1.2 Web socket communication protocol

The web socket communication has to achieve the same goal as the Bluetooth communication being transmitting matrix A and B. Depending on the efficiency and performance of this web socket communication we set up a similar set of rules. Compared to Bluetooth, the web socket transmission is much more efficient. We can easily transmit over eight characters without having any data loss. This means that we do not have to send acknowledgement messages like the "O" or "R" message that were necessary with the Bluetooth communication in order to have a solid data transmission.

If we look at subsection **??** under data frames, we see that the payload length of a message can be either 16 or 64 bits. But this does not mean we are limited to sending messages smaller than 64 bits at a time since the socket will just split up the data in seperate messages. This allows us to basically have any message size. Because of limited memory we chose to use integer numbers between 0-99. Next, we decided to combine 4 numbers and make one message with them. The message now contains 4 integers (16 byte). We convert the integers to strings and add a "0"-string in front of numbers that are smaller than 10 to make sure the full message has a constant length of eight characters.

In order to have any communication at all, we start of by connecting both devices through web socket technology. As soon as both devices are connected with each other, the SoC will again start of by sending a message. The message will ask the phone to send matrix A. The phone then will start transferring all data from matrix A to the SoC. Whenever matrix A is transmitted completely, the SoC will ask to transfer matrix B and the phone will send matrix B. After the last number of matrix B being transferred, the SoC will confirm that all data has been received. In figure 2.5 we can see the exact messages needed from the SoC in order to have a correct communication process. The messages from the phone to the SoC are examples of how the messages could look like. The first message "12345678" would be numbers 12, 34, 56 and 78 being transmitted to the SoC.

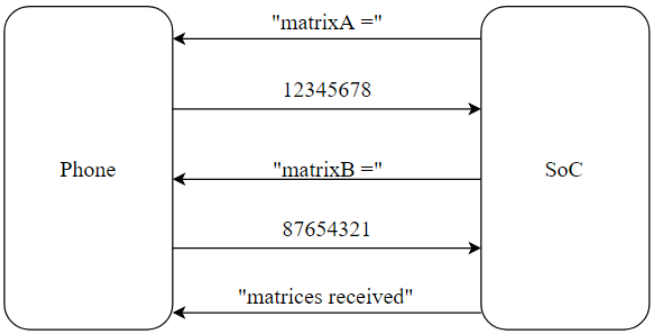


Figure 2.5: Sending matrix A and B

After the SoC sends its last message it will start doing the matrix multiplication of matrix A and B with the outcome being matrixC. When the SoC is ready it will send the message "matrixR =" to notify the phone it will start sending over the values within matrix C. Unlike the phone, the SoC will send each number one by one since the number length varies with the size of the matrix. Figure 2.6 shows both types of messages being send by the SoC. The phone should receive all numbers without having to give any feedback to the SoC. Whenever the full result matrix is transmitted by the SoC, the web socket connection is terminated and the complete process is finished.

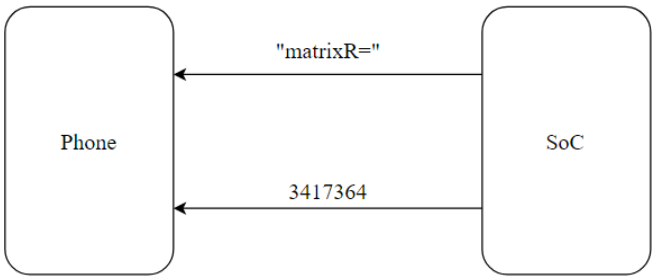


Figure 2.6: Sending result matrix

2.2 Android apps

The android phone needs proper software in order to function well in our expirement. To build this software we use Android Studio. It is an easy-to-use java environment with a lot of information that can be found on the internet. We will make three different applications using Android Studio. One application to test the matrix mulitplication with the phone itself to compare the speed with the SoC. A second application is responsible for the Bluetooth communication and a third application is used for the web socket communication.

2.2.1 Matrix multiplication

This app will multiply two square matrices that are filled with random integers converted to strings (see subsection 2.1.2 on why we do this). We start of with creating two matrices. Listing 2.1 shows how this is done. We start of by making an array of integers that is "arraySize" large. We start a for loop that will cycle "arraySize" times. In line 4 we create a random integer between value 0 and 99 and put this in the array "matrixA" in the next line. MatrixB is made in exactly the same way.

We then go through listing 2.2 which is a basic way of multiplying two matrices.

```
1| matrixA = new Integer[arraySize];
2| int random;
3| for (int i=0; i<arraySize;i++){
4|   random = (int) (Math.random()*100);
5|   matrixA[i] = random;
6| }
```

Listing 2.1: Creating matrix

```
1| t1 = (int) System.currentTimeMillis();
2| for(int i=0; i<arraySize; i++){
3|   for(int j=0; j<arraySize; j++){
4|     sum = 0;
5|     for(int k=0; k<arraySize; k++){
6|       sum = sum + matrixA[i*arraySize+k]*matrixB[k*arraySize+j];
7|     }
8|     matrixC[i*arraySize+j] = sum;
9|   }
10| }
11| t2 = (int) System.currentTimeMillis();
```

Listing 2.2: Code for matrix multiplication on Android phone

First, we determine the system time and assign this value to t1. In section **??**, we explain how matrix multiplication works so this should clear up lines 2-10 in the listing above. After the multiplication is finished we determine another system time and assign this value to t2. If we distract t1 from t2, we know how many milliseconds it took to complete the full multiplication. The layout of this app can be seen in figure 2.7.

2.2.2 Bluetooth application

In the next two apps, we will transfer the matrics over to the SoC that will perform the matrix multiplicaiton. First, we'll explain how the Bluetooth app works.
Since this app works with bluetooth, we have to set up the permissions for the app to use the phone's Bluetooth adapter in the AndroidManifest.xml file like so:

<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>

The first thing the app will do is check wether or not your phone's Bluetooth

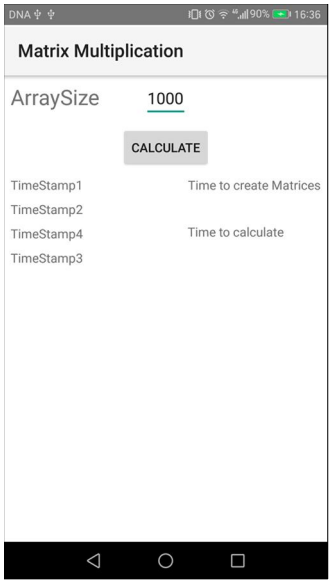


Figure 2.7: Matrix multiplication app

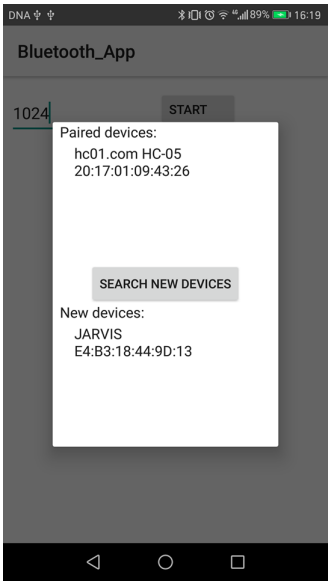


Figure 2.8: Bluetooth menu

is enabled. If not, it will ask you if you want to enable the Bluetooth. If you press allow, you'll be directed to the main screen. If the Bluetooth was already enabled, nothing will pop up and you'll see the main screen in front of you. The front end of the app is simple, consisting of only one textview and two buttons. The textview has the purpose to allow you to input a custom size for your matrix's row and column. The first button "Start" will make 2 matrices with the given size. The other button will open a menu with 2 lists and a third button. The first list will show you all the already paired Bluetooth devices on your phone. The other list is empty, but after pressing the button "Search new devices", the list will fill up with discoverable devices around you as you can see in figure 2.8.Tapping on one of the new devices will result in an attempt to pair with this device. Tapping on a paired device in the list will result in an attempt to connect to that device. The device you selected will be the device you will communicate with. Android Studio provides libraries that easily gives you acces to the Bluetooth adapter in the phone. More information on how the Bluetooth adapter works can be found on the Android developer website [13]. When the connection is established, the protocol described in 2.1.1 will be executed.
The Bluetooth adapter has an in- and an outputstream to respectively read and write data. Both of which run in a thread so whenever there is new data coming from the SoC, the app will be able to read that. The received messages go to the message handler. In this message handler, various things happen depending on which message was received. Some of the effects of

the messages are described in 2.1.1 and here we'll explain what happens in the background of the app.
Listing 2.3 is a piece of the handler code. First, we set up some byte array variables. One of these will read and copy whatever is in the buffer, while the other is used to make our messages send to the SoC. The following if-statement asks if the phone is the device that is currently "sending" matrices. If so, a second if-statement awaits with cases for all possible messages the phone can receive from the SoC. As we already know, when the phone receives the message "C", we will be starting to send over the matrices. We define an integer "i" that functions as an index to go through the matrix that contains the messages we have to send. We define a new message with the first element of the array and we define a first timestamp that indicates the beginning of the whole process. Next up is the "O" message. This message is received whenever the SoC acknowledges that it has received our previous data. We increment our index and check if the index is not exceeding our matrix length. If the index is ok, we send a message with the next array component. When the SoC didn't receive our message correctly, we should receive the message "R". If this is the case we will not increment the index so it will just send the previous message again. As for now, we only send one matrix over to the SoC for the sole purpose of testing the transmission time since sending the second matrix would take the same amount of time because they have the same size. Whenever the matrix has been transfer, the SoC will send an "X". At that point we make another timestamp so we know the elapsed time for sending one matrix from the phone to the SoC. The time will be printed in the log with the code shown in listing 2.4. In the meanwhile, the phone sends a message "C" to the SoC in order to switch the roles. The SoC will become the transmitter and the phone will act as a receiver.

Listing 2.5 starts of with the else-statement that is related to the if-statement from listing 2.3. Now that the phone is the receiver the first message we would receive from the SoC should be a "D", indicating the SoC is done with the matrix multiplication. We send over message "C" to tell the SoC that the phone is ready to receive data. We also set up a timestamp to indicate the beginning of the receive time. Further messages that will be received will be a set of numbers with a ":"-symbol at the end to indicate that it is the end of the message. In subsection 2.1.1 we explain why we do this. If the received number follows the rules of the protocol we confirm that we received the number correctly by sending an "O" as a message to the SoC. In case we suffered from data loss or any other error, we send an "R". Line 10 in listing 2.5 will check wether or not our matrix is full. If that is the case we make a second timestamp and print the time it took to receive te matrix.

```
1| byte[] readBuf = (byte[]) msg.obj;
2| byte[] newMsg;
3| String readMessage = new String(readBuf, 0, msg.arg1)
4| if(sending){
5|   if (readMessage.equals("C")) {
6|     i = 0;
7|     newMsg = (String.valueOf(matrixD[i]) + ":").getBytes();
8|     t1 = System.currentTimeMillis();
9|   }else if (readMessage.equals("O")) {
10|    i++;
11|    if(i<matrixD.length){
12|      newMsg = (String.valueOf(matrixD[i]) + ":").getBytes();
13|    }
14|  }else if (readMessage.equals("R")) {
15|    newMsg = (String.valueOf(matrixD[i]) + ":").getBytes();
16|  }else if (readMessage.equals("X")) {
17|    t2 = System.currentTimeMillis();
18|    newMsg = "C".getBytes();
19|    printTime();
20|    i = 0;
21|  }
22|  else{
23|    newMsg = ":".getBytes();
24|  }
25| mChatService.write(newMsg);
```

Listing 2.3: Bluetooth message handler as transmitter

2.2.3 WebSocket application

Just like in the Bluetooth application, the SoC has to solve the matrix multiplication. Again, the front end of the app is really simple, containing only a textview and a button (figure 2.7). The textview is for defining a matrix size for the rows and columns. With the first tap on the button, you confirm the matrix size and create 2 square matrices with the chosen size containing random values from 0 to 99. The second tap will connect you to the WebSocket whos IP address is predefined as you can see in line 4 of listing 2.6. In order for us to connect the phone to the WebSocket server, we need to create a WebSocket client first. The WebSocket client can be created with the code shown in listing 2.6. For the WebSocket client, we used the "org.java-websocket:Java-WebSocket:1.3.0" library since it is good and easy to use. Also, to allow the app to use the phone's internet connection, we have to put the following line in the AndroidManifest.xml file:

```
1| public void printTime(){
2|     Log.i(TAG, "------------------------------------------------------");
3|     Log.i(TAG, "Time = " + String.valueOf(t2 - t1));
4|     Log.i(TAG, "------------------------------------------------------");
5| }
```

Listing 2.4: Prints elapsed time for sending matrices

<uses-permission android:name="android.permission.INTERNET"/>

Now, whenever we call the method to initiate the WebSocket client, the URI from the server is required (line 4). Line 9-50 are shown in listing 2.7 that gives the detailed code on which methods the socket has to contain in order to function properly. In line 51 we try to establish a connection between the client and the server.
The socket requires certain methods that are invoked on a call from the server. For example, if the client connects succesfully with the server, method "OnOpen" is invoked. In this method we print a message to our Log that the phone succesfully connected to the server. The second method required for the WebSocket client is "onMessage". This method is invoked whenever the client receives a message from the server. What happens next depends on the message this method received. All different types of messages are put into an if-statement so we execute different code depending on the received message. You can read more about the messages on line 16, 19 and 23 in subsection 2.1.2 where the transmission protocol is explained. If we receive "matrixA=" or "MatrixB=", we respectively start transferring matrix A and matrix B. But before we start sending them over, we create a timestamp t1 that will measure the system time in milliseconds at that point. We do the same thing for t2 as soon as we start sending matrixB. Having a timestamp for matrixB seems redunant but why this is done will be explained in the results. A third timestamp t3 is defined after receiving the message "matrixR=" which will represents the start of receiving data. The message in the next case is to determing how long the SoC's execution time was over calculating the matrix multiplication. When we receive message "done :D", we measure out last timestamp and close our WebSocket connection.

```
 1| else{
 2|   if(readMessage.equals("D")){
 3|     newMsg = "C".getBytes();
 4|     t1 = System.currentTimeMillis();
 5|   }else if(readMessage.length()==wordSize+1 && readMessage.endsWith("
 6|      try{
 7|       matrixC[counter] = Integer.parseInt(readMessage.substring(0, wo
 8|       counter++;
 9|      }catch(Exception e){}
10|       if(matrixC[elements-1] != null){
11|         t2 = System.currentTimeMillis();
12|         printTime();
13|       }
14|       newMsg = "O".getBytes();
15|   }
16|   else{
17|     newMsg = "R".getBytes();
18|   }
19|   mChatService.write(newMsg);
20| }
```

Listing 2.5: Bluetooth message handler as receiver

```
 1| public void connectWebSocket() {
 2|   URI uri;
 3| try {
 4|   uri = new URI("ws://130.230.144.40:8080");
 5| } catch (URISyntaxException e) {
 6|   e.printStackTrace();
 7|   return;
 8| }
 9|   socket = new WebSocketClient(uri, new Draft_17()) {...};
51|   socket.connect();
52| }
```

Listing 2.6: Making a WebSocket client

```
 9| socket = new WebSocketClient(uri, new Draft_17()) {
10|   @Override
11|   public void onOpen(ServerHandshake serverHandshake) {
12|     Log.i(TAG, "Connected to: " + uri.getHost() + ":" + uri.getPort()
13|   }
14|   @Override
15|   public void onMessage(String s) {
16|     if(s.equals("matrixA=")){
17|       t1 = System.currentTimeMillis();
18|       sendMatrixA();
19|     }
20|     else if(s.equals("matrixB=")){
21|       t2 = System.currentTimeMillis();
22|       sendMatrixB();
23|     }
24|     else if(s.equals("matrixR=")){
25|       t3 = System.currentTimeMillis();
26|     }
27|     else if(s.contains("Execution time in")){
28|       exeTime=s;
29|     }
30|     else if(s.equals("done :D")){
31|       t4 = System.currentTimeMillis();
32|       displayTimes();
33|     }
34|     else{
35|       try{
36|         int i = Integer.parseInt(s);
37|         matrixC[counter] = i;
38|         counter++;
39|       } catch(Exception e){}
40|     }
41|   }
42|   @Override
43|   public void onClose(int i, String s, boolean b) {
44|     counter = 0;
45|   }
46|   @Override
47|   public void onError(Exception e) {
48|     //Log.i(TAG, "Error " + e.getMessage());
49|   }
50| };
```

Listing 2.7: Detailed look at creating the WebSocket client

Chapter 3

Results

3.1 Phone performance

In this section we will describe the performance and the results we got from our applications from section 2.2. The phone we used is a Huawei P9 (EVA-L09) that has following specifications:

- CPU: HiSilicon Kirin 955 2.52 GHz
- RAM: 2780MB
- Android version: 7.0

3.1.1 Matrix multiplication

The Huawei P9 is quite a powerful device and was able to outperform the SoC when calculating with medium sized matrices. When the app is executed correctly the screen should look like figure 3.1. While testing, we came accros some interesting findings when calculating matrices with the size of 1024. The time it took skyrockets when the matrix has an exact size of 1024. One column/row more or less and the app would behave as expected. From there on, the time for certain matrix sizes is inconsistent. Because we work with large matrices, we can hardly determine wether or not the phone is executing everything correctly making the results in figure 3.3 for large matrices unreliable.
Figure 3.2 shows us the CPU and RAM usage over time. In the RAM graph we can clearly see where the matrices are generated. Over time the RAM usage increases and it drops after all matrices in the app are cleared. The CPU usage is around 12-13% throughout the whole process.

3.1.2 Bluetooth app

After connecting the devices, the SoC will request for the transmission of the data. There is not a lot happening in the front end but the transmission

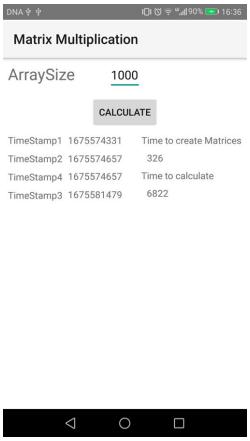


Figure 3.1: Result screen matrix multiplication

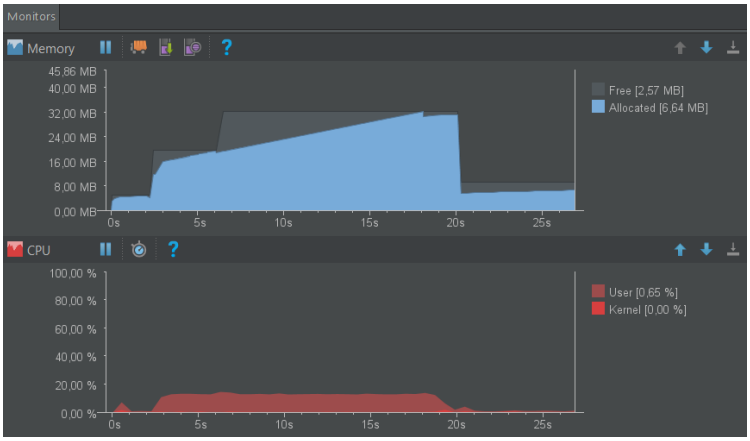


Figure 3.2: CPU and RAM usage

time and receive time are displayed in the Logcat like in figure 3.5 when the full process is done. Again we will check out the CPU and RAM usage of the app as we did with the previous app. If we look at figure 3.4 we notice a lot of changes. The app barely uses and CPU power and also only requires a small amount of RAM to create the matrices. As for the Bluetooth app we were only able to test matrices that were smaller than 20x20 as the SoC would give the error: "Too many files open".

3.1.3 WebSocket app

As for the WebSocket, there is a little more interesting data to show. If we look at the RAM usage in figure 3.7, the first increase in RAM is the app startup. The next one is where we create the matrices. Around 8 seconds the matrices are created and the network graph indicates we are transmitting the matrices over to the SoC. The decrease around 18 seconds indicates that we clear the matrices that have been send. After that we are in an idle state where we sent the last few messages that are in the WebSocket queue. When the SoC executed the multiplication it will send the solution back to the phone, hence the receive (Rx) spikes in the network graph. When we look at the CPU, it is mostly used when we receive data from the SoC. This is probably CPU intensive since we slowly fill up the solution matrix with the data that is received from the SoC.
The log file in figure 3.6 shows all time intervals mentioned in 2.2.3.

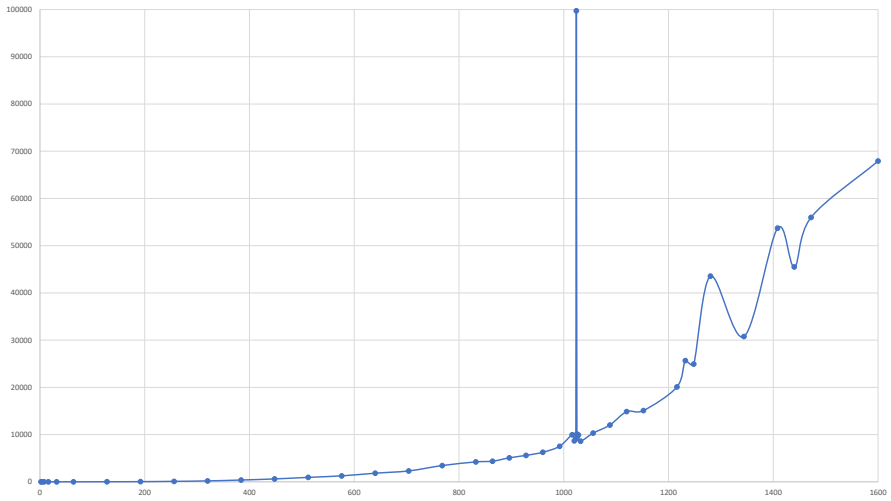


Figure 3.3: Results matrix multiplication

3.2 Bluetooth vs web socket communication speed

In order to choose the best communication protocol, we have to test both Bluetooth and WebSocket communication preferable with large sized arrays. First, we perform the actual communication in order to find a relation between the arraysize and the elapsed time. When we find this relation we can set up a formula that allows us to calculate an estimate value for the elapsed time for any matrix size.

3.2.1 WebSocket transmission

We first performed multiple calculations with WebSocket communication. We seperated the elapsed time it took the phone to transmit the data and the elapsed time it took the SoC to transmit the data. The results in the spreadsheet from appendix **??** represent the elapsed time for sending, receiving and the total time. If we divide the total number of seperate symbols in one matrix with the elapsed time we know the amount of symbols that are transmitted per second (sym/s). For small matrices, the sym/s is not reliable since there are background processes that influence these results. Although, these background processes barely affect the sym/s with larger matrices. So in order to determine the relation, we will take an average of all the sym/s with all matrix sizes, apart from those that are highly affected by background processes. This resulted in a sym/s value of 86843,61 for the phone transmission and 127228,7 for the SoC transmission. Knowing these values we can make graph 3.8. In the graph we compare the transmission time for the phone with the time for the SoC. The small dent in the SoC curve is the change in average number length in the matrix from 6 to 7.

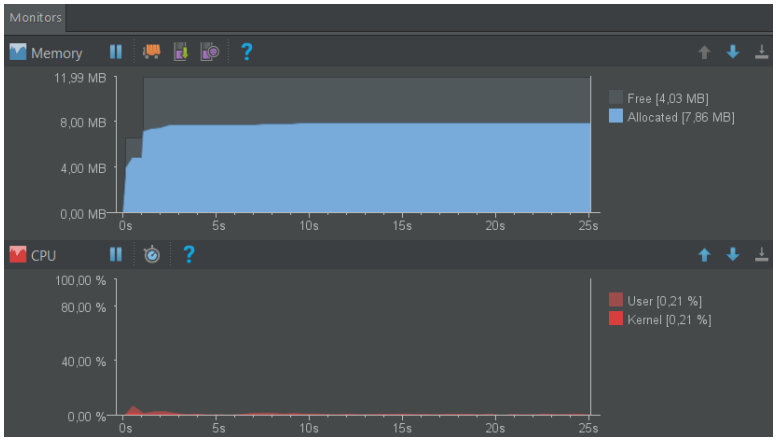


Figure 3.4: CPU and RAM usage Blueooth

```
06-02 04:14:14.641 I/BluetoothChatFragment: ------------------------------------------------
06-02 04:14:14.645 I/BluetoothChatFragment: Transfer time = 2583
06-02 04:14:14.647 I/BluetoothChatFragment: ------------------------------------------------
06-02 04:16:24.855 I/BluetoothChatFragment: ------------------------------------------------
06-02 04:16:24.860 I/BluetoothChatFragment: Receive time = 2693
06-02 04:16:24.863 I/BluetoothChatFragment: ------------------------------------------------
```

Figure 3.5: Bluetooth app log

3.2.2 Bluetooth transmission

Next we determine the transmission time using the Bluetooth communication. We only mention the practical elapsed time to send over one matrix from the phone to the SoC the elapsed time the other way around was very similar. We were only able to test small matrices of up to a size of 20x20. The sym/s for this communication protocol was around 258 sym/s including a lot of repeated messages due to package loss. Although we are not able to transfer large matrices, we are able to set up a formula that allows us to make an estimate of the elapsed time for higher matrices. The results for this are shown in graph 3.9.

```
06-01 14:50:58.446 I/ContentValues: Connected to: 130.230.144.40:8080
06-01 14:51:31.756 I/ContentValues: ------------------------------------------------
06-01 14:51:31.756 I/ContentValues: Transfer time =         19894 ms
06-01 14:51:31.756 I/ContentValues: Execution time in milliseconds = 1981.023 ms
06-01 14:51:31.756 I/ContentValues: Receive time =      12370 ms
06-01 14:51:31.756 I/ContentValues: Total time =       32974 ms
06-01 14:51:31.756 I/ContentValues: ------------------------------------------------
```

Figure 3.6: WebSocket app log

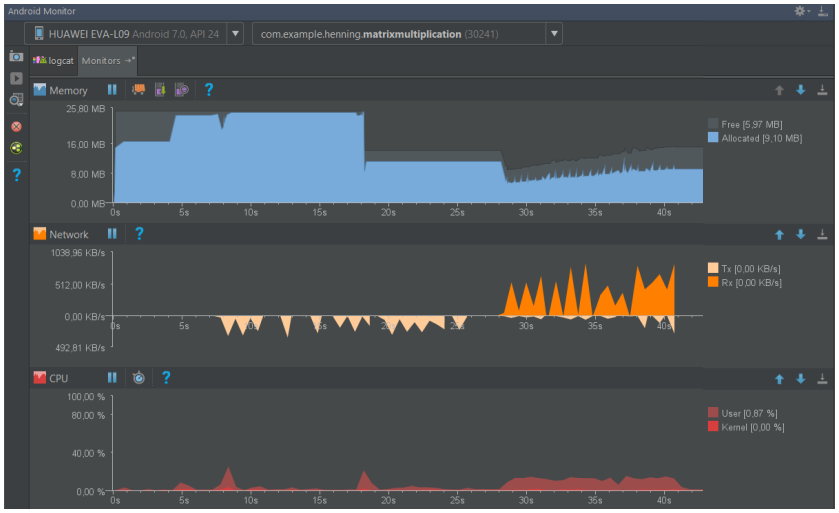


Figure 3.7: CPU, RAM and network usage WebSocket

3.2.3 Bluetooth vs. WebSocket

Comparing the s/c and looking at both graph scales, it is very clear that the WebSocket is way more faster than Bluetooth communication. The WebSocket transfer time from the phone to the SoC is around 340 times faster while vice versa the WebSocket is almost 500 times faster. The WebSocket also allowed us to practically transfer much larger matrices aswell making it superior compared to our Bluetooth communication. Graph 3.10 visualizes the difference in scale for both communication protocols. We can clearly see that the WebSocket requires less time to transfer one matrix compared to Bluetooth.

3.2.4 WebSocket and SoC vs. Android smartphone

As discussed in previous subsection, we can conclude that there is no way of Bluetooth being worthwhile for this project. Thus, we are only testing the WebSocket protocol combined with the SoC's ability to multiply matrices. Subsection **??** gives us a comparison between the phone and the SoC on how fast they can process the matrix multiplication. Now we want to know at which point the Android phone will be slower than transferring all three matrices and performing the matrix multiplication on the SoC. If we look back at graph figure **??**, the orange curve represents the estimate of the total transfer time of all three matrices. While the yellow and gray curves respectively represent the calculation time it takes to perform the multiplication with the SoC and the Mac. The yellow and gray curves are trendlines derived from data we achieved from performing multiple matrix multiplications with different matrix sizes. The equation for the yellow curve

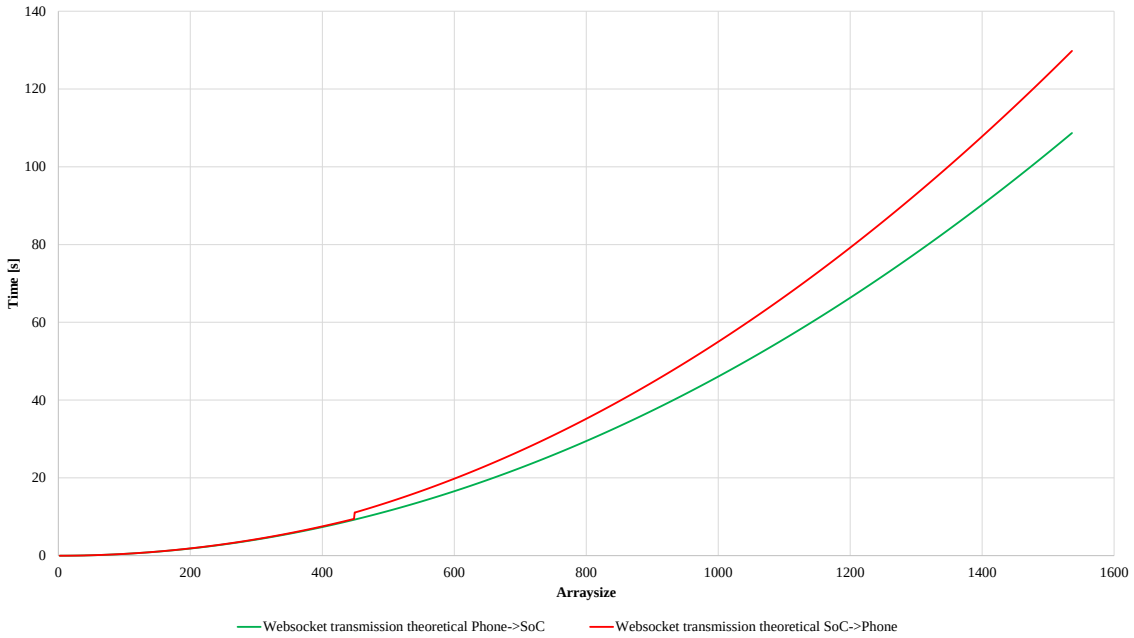


Figure 3.8: WebSocket transmission time for phone to SoC and vice versa

trendline is equation **??** and the grey curve's trendline is equation **??**. Combining these gives us the blue curve which represents the combination of the transfer time and the Mac calculation time, while the green curve shows the combination of the transfer time and the SoC calculation time. If we look at the green curve we see it crosses the black curve, which is the trendline made with equation **??** around a matrix size value of 2350. If we want a more correct value for this we take a look at the equation that defines both curves. Equation **??** represents the used function to generate the trend line of the Android phone execution time. Equation 3.2 is a combination of the data transfer time from both the phone and the SoC. The transfer speed from the phone is 43,42 numbers/second while the SoC can transfer the larger numbers back to the phone at a rate of 18,85 numbers/second. In order to find the the value for the matrix size where the phone execution time will be the same as the full process time from the SoC, we have to find x in y=v+w, with y equal to equation **??**.

$$v = -3 * 10^{-10} * x^4 + 2 * 10^{-5} * x^3 - 0,0021 * x^2 - 0,2801 * x + 142,83 \quad (3.1)$$

$$w = \frac{2 * x^2}{43,42} + \frac{x^2}{18,85} \quad (3.2)$$

if we extract x from y=v+w we get a value of 2338. This means we have

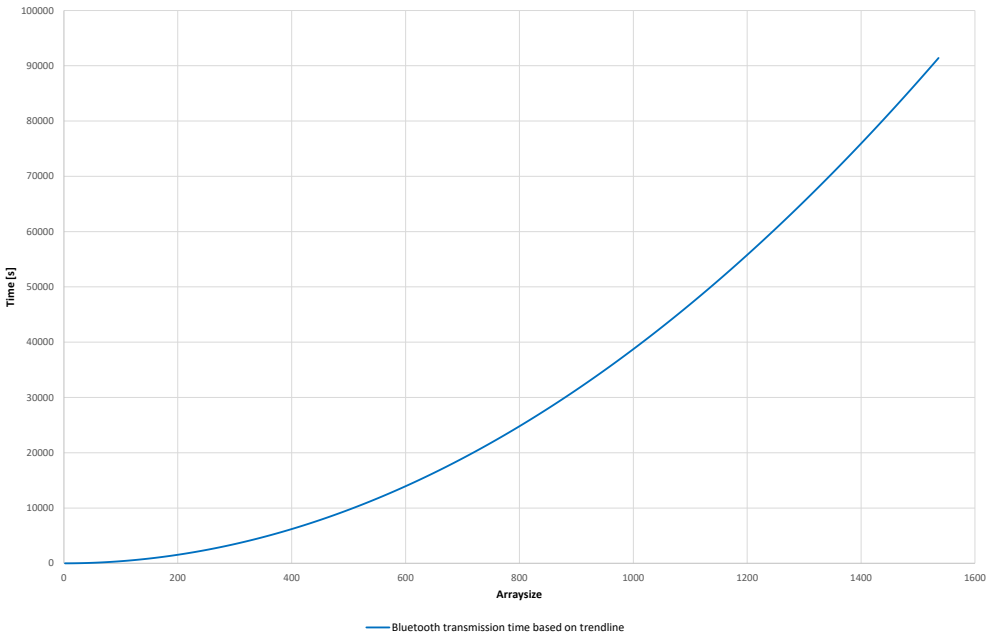


Figure 3.9: Bluetooth transmission time

an even time for both processes in case we try to do a matrix multiplication with two 2338x2338 matrices.

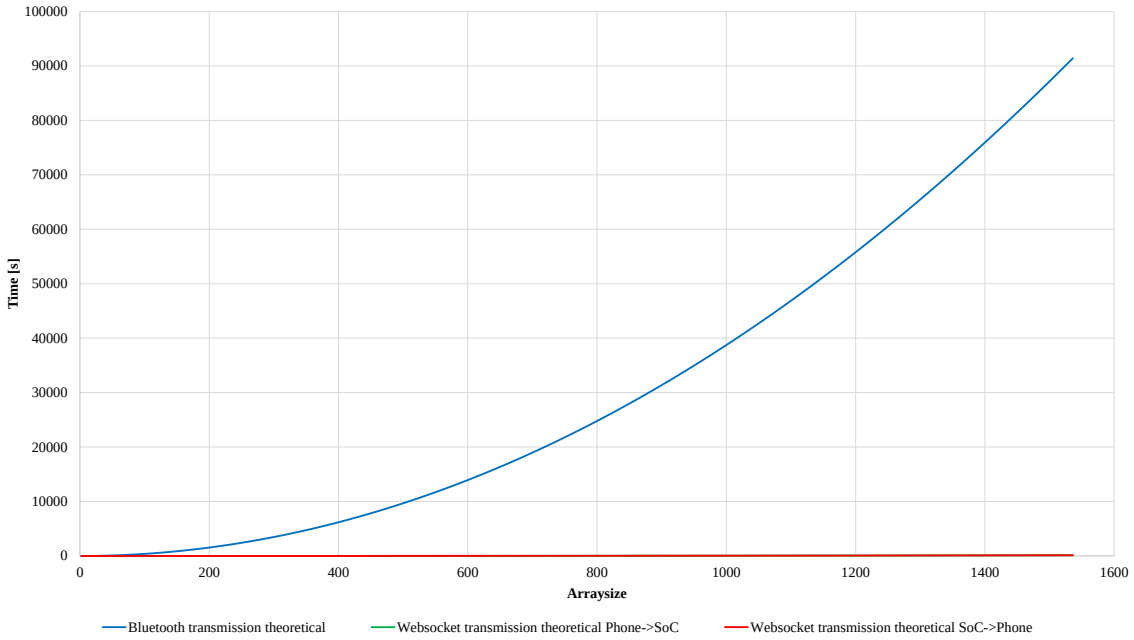


Figure 3.10: Bluetooth transfer time vs. WebSocket transfer time

Matrix size	Send time [ms]	Receive time [ms]	Total time [ms]
2	11	0,67	443
4	46	4	496
8	82	7	549
16	79	44	585
32	75	81	620
64	220	189	840
128	796	806	1964
192	1578	2101	4332
256	2800	3508	6945
320	4340	5662	10943
384	6671	8396	16345
448	8997	11777	22426
512	11027	13935	27715
576	14422	17941	35947
640	17713	23354	45721
704	21311	26985	54404
768	25133	32571	65408
832	39736	36577	76310
896	47000	41384	88408
960	54339	51809	106180
1024	62391	58079	120531
1088	60050	63463	136234
1152	57709	68846	151937
1216	64762	78557	172228
1280	69590	82446	189864
1344	77666	88812	206451
1408	86274	100357	233396
1472	91222	116673	261685
1536	102046	113719	278859
1664	255422	143429	398862
2048	179112	230696	558275

Bibliography

[1] B. Barney, "What is parallel computing," *Introduction to Parallel Computing*, 2012.

[2] *MCSE-011: Parallel Computing.* Gullybaba Publishing House, 2008.

[3] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[4] [Online]. Available: https://www.khronos.org/members/list

[5] S. M. Patterson, "Q1 2017 smartphone shipments: Samsung rebounds, apple goes sideways, chinese makers roar," 2017.

[6] J. Vincent, "99.6 percent of new smartphones run android or ios," *The Verge*, 2017.

[7] T. Saggu, "Bluetooth technology," *S.U.S.CET.*

[8] W. Stallings. (2001) Introduction to bluetooth. [Online]. Available: http://www.informit.com/articles/article.aspx?p=23760

[9] Q. H. Mahmoud. (2003) Wireless application programming with j2me and bluetooth. [Online]. Available: http://www.oracle.com/technetwork/systems/index-156651.html

[10] A. Melnikov. (2011) The websocket protocol. [Online]. Available: https://tools.ietf.org/html/rfc6455

[11] Mozilla. (2017) Writing websocket servers. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

[12] J. Lawson. (2012) Websockets, a guide. [Online]. Available: http://buildnewgames.com/websockets/

[13] Bluetoothadapter. [Online]. Available: https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html