The Creation and Implementation of a Reliable Low Latency Communication System for Mediated Social Touch in Esports

A Future Wearables Proof of Concept

Bachelor's thesis
Degree Programme in Computer Applications
Autumn 2024
Brian Scotcher

HAMK
Häme University
of Applied Sciences

Degree Programme in Computer Applications or BIT — Abstract
Author Brian Scotcher — Year 2024
Subject The Creation and Implementation of a Reliable Low Latency Communication System for Mediated Social Touch in Esports
Supervisors Esa Huiskonen

The purpose of this thesis was to document the software creation and implementation for Spiritus Ludi – a smart sleeve platform for esports players in remote settings. Conventional touch in sports has been extensively researched and has found overall that encouraging gestures such as high fives and fist bumps improve a team's emotional state, cohesion, and other factors, which have been found to influence overall performance. Spiritus Ludi was created to research the effects of social mediated touch in teams who are in remote settings. In this case, esports teams are the focus. This multi-disciplinary project required expertise from numerous fields, including the creation of software to enable the Spiritus Ludi sleeves to send and receive data on wearable technology in real time. The overall Spiritus Ludi project was led by Satu Jumisko-Pyykkö at HAMK Smart.

To create the appropriate solution, this thesis identifies and analyses existing forms of communications in esports and the associated forms of data transports protocols and characteristics. These are primarily identified as TCP and UDP protocols. The thesis then proceeds to outline what Spiritus Ludi is in detail, followed by an overview of the Bluetooth Low Energy and SignalR technologies. Next, the purpose of the Spiritus Ludi software is outlined along with the required characteristics. The characteristics identified as high reliability, low latency, and sustainable and expandable are explained in relation to why this is necessary for communication an esports team. This thesis then discusses the development of the software. It is broken down into three parts, identified as the Spiritus Ludi Firmware, the Spiritus Ludi Software, and Spiritus Ludi Real Time Communication, which utilises the cloud technology. The code is shown with accompanying explanations of how each section works and the software packages and accompanying versions used.

The outcome of the Spiritus Ludi software culminated in the successful field tests conducted by the Future Wearable researchers. While it was acknowledged that this implementation is a prototype, the overall success of the Spiritus Ludi project has led to actions being taken for further development. Recommendations include further research and development into firmware updating, BLE connection security, and testing in future iterations.

Keywords WebSocket, Bluetooth Low Energy, React Native, Arduino, C#
Pages 43 pages and appendices 1 page

Glossary

BLE	Bluetooth Low Energy
CLI	Command Line Interface
CS:GO	Counter Strike: Global Offensive
DFU	Device Firmware Update
GATT	General Attribute Profile
HTTP/2	Hypertext Transfer Protocol Version 2
iOS	iPhone Operating System
I2C	Inter-Integrated Circuit Protocol
LED	Light Emitting Diode
MAC	Media Access Control
MITM	Man in the Middle
OTA	Over the Air
RTC	Remote Procedure Calls
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WebRTC	Web Real Time Communication
WGSN	Worth Global Style Network

Content

Figures and tables

Appendices

Appendix 1. Material Management Plan

1 Introduction

The popularity of esports has grown drastically in the last 10 years. Along with this, the industry itself has also grown with it. Millions of U.S Dollars have been invested into esports, and it has proven to be a lucrative industry on a global scale (Bousquet & Ertz, 2021, pp. 9–13). The technology involved in esports has also seen major advances over the same period, with companies such as Intel, AMD, and Nvidia delivering leading technology which benefit many people and industries, including online gaming and esports. One important aspect within esports teams, especially at a professional level is clear and reliable communication. Different systems have been developed by esports teams and companies to assist with in-game communication. These systems are a mix of technological solutions and agreed communication standards set within a team. Mediated social touch is the replication of touch through technology. Devices like this exist, however at the time of writing this thesis, there are no released products that are created for the context of esports. The use and effects of mediated social touch in esports have now been explored with the Spiritus Ludi Project (Spiritus Ludi - Spirit of the Game in Esports - HAMK - Häme University of Applied Sciences, n.d.).

This thesis will address the software requirements of what such a system would need to meet in the context of mediated social touch in esports, the application of the different software systems and libraries used to create such a system, and the documentation of the created system, which met the needs of Spiritus Ludi – the project that encompassed this thesis and its associated work. Issues found during development will be documented, as well as references to the data collected. To conclude the resulting performance of the software and suggest potential further development and areas of research will be discussed. The associated software to this thesis was created for HAMK Smart. Spiritus Ludi was created from collaboration between HAMK Smart and TUNI within the “Developing Competence in Smart Wearables” project.

Due to the multi-disciplinary nature of Spiritus Ludi, it is important to state the boundaries of what this thesis will address. The primary focus of this thesis is the backend software systems associated with real time communication, the back-end software developed on for connected clients, and a limited overview of the client mobile device - wearable device communication. There will also be a general technical overview of the end-to-end communication. This thesis will not address the front-end UI design process or design iterations relating to the creation of the Spiritus Ludi device. Additionally, it will not include detailed document of the associated hardware, however there will be references to some of the components used as this has impacted on the software implementation.

This thesis aims to answer the following questions:

1: What back-end software is currently available to convey mediated social touch?

2: How can this software be applied to build a mediated social touch system for esports?

3: Why do mediated social touch systems need to be reliable in the context of esports?

2 Communication Systems in Esports

Most of the communication between team members in esports is conducted through audio and text dialogue, with the preference for games such as CS:GO being audio (Oksala, 2022). The effectiveness of this communication is dictated by numerous factors. Some of these are the communication skill level of individuals, the time teams put into practice communicating with each other, and the limitations of the game they are playing.

One communication method that is not currently utilised in esports is touch. Touch in conventional team sports such as football and basketball has been proven to promote positive feelings, team cohesion, and improve overall performance. In round based competitive esports games, touch may not be appropriate between team members due to the small amount of time available between rounds, or in remote team set ups where touch is not possible due to the geographical locations of players. Esports players maybe be able to use touch if they are in a co-located setting, however often they are not and therefore communicative touch gestures is not an option for many players.

2.1 Existing Communications Systems in Esports

The primary method of communication used between players in esports teams are text and verbal. Most games that are competitively played in esports come with integrated communication systems that allow players in a team to communicate. These systems tend to be integrated into the game itself or into the platform that is hosting the game. Communication is also regulated in competitions to protect the integrity of the competition.

However, adding such functionalities to computer games poses its own challenges. Video game developers need to find the balance between providing a clear, usable, and reliable communications system without infringing on the flow of the designed game or causing a distraction that it would impede the players performance. One study found that realistic sound was considered the most important feature among a group of video game players (Wood et al., 2004). Incorrect or poor team communication systems in online games can also have a large impact on the success of a video game. In software development additional complexity increases the risk of bugs and faults in software. This is also applicable in game development. A game can be rendered unplayable if the integrated chat client is causing the game to crash.

A popular communications technology utilised by game developers and platforms for its in-team communication is WebRTC.

2.1.1 User Datagram Protocol

WebRTC stands for Web Real-Time Communication. It uses various protocols to enable audio and video communication without requiring any plug-in installations or third-party applications. (WebRTC API - Web APIs | MDN, 2023).

The preferred communication protocol for WebRTC is the User Datagram Protocol (UDP). Created in 1980, UDP is a low latency Transport Layer Protocol, making it ideal for services such as video and audio streaming. UDP will send data to the target computer without establishing a formal connection to that computer. Data packets that were not received by the recipient will not be sent again, therefore allowing for potential data loss. There is no way in UDP to confirm that the data has been received by the target computer. Audio and video streaming can have a degree of data loss tolerance, making UDP a suitable choice for WebRTC video and audio streaming.

2.1.2 Transmission Control Protocol

WebRTC can also use Transmission Control Protocol (TCP). Unlike UDP, TCP is a reliable, in order, byte stream service to applications (RFC 9293: Transmission Control Protocol (TCP), n.d.). TCP checks that the data received is complete using per segment checksum values and will resend missing segments to ensure that the full transmission of data is complete. The drawback of this is that it is slow as there is a constant Request/Respond process happening between the server and client. This makes TCP ideal for text chat between individuals and groups. However, the connection is not kept open, meaning new connection needs to be established with every request.

2.1.3 Communication Protocols Suitable for Mediated Touch in Esports

TCP and UDP both have their strengths depending on the requirements of communication needed. Today they are acknowledged as two common core protocols of internet communication. However, UDP connections do not have the required characteristics of low latency data safe transport needed for a social mediated touch platform due to its data loss tolerance. Single use TCP which is the conventional use where a single request is made, responded to, and upon successful transfer of all the data the connection is closed has the desired data safe transportation requirements. For the needs of a social mediated touch system, a TCP connection would require persistence, which is what HTTP Version 2.0 (HTTP/2) provides. Repetitively closing and opening connections can result in excessive latency, creating unnecessary network traffic. HTTP/2 connections are persistent by design (Belshe et al., 2015). Implementing a WebSocket protocol layered on top of a

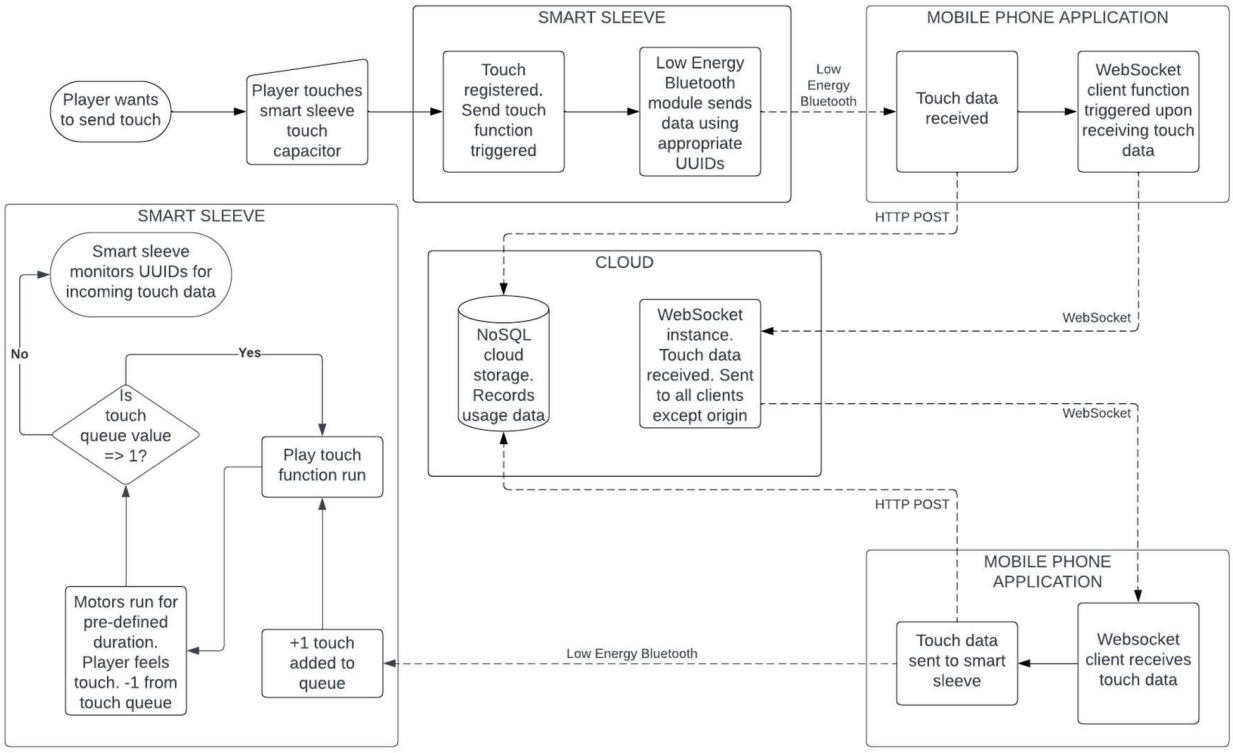
single persistent TCP connection can provide simultaneous bi-directional communication (Melnikov & Fette, 2011).

2.2 The Spiritus Ludi Sleeves and how they work

The Spiritus Ludi sleeves are designed to be used by esports teams, usually consisting of approximately six players. During the flow of their chosen game – in the case of the field tests, Counter Strike: Global Offensive (CS:GO), the players will send digital high-fives to each other as a form of positive encouraging interaction. Anonymous usage statistics are taken and used in conjunction with other data gathering methods by the researchers to examine user behavior and emotional state when using the Spiritus Ludi system.

Figure 1 shows the actions which occur when a user sends a digital high-five from their sleeve. The sleeve will register the gesture and use a Bluetooth connection to transfer that to a paired mobile application created specifically for the Spiritus Ludi sleeve. The mobile application then uses WebSocket technology to transfer that data to a cloud hosted WebSocket Hub instance where it is then distributed to all other clients. Asynchronously using a parallel connection to Azure, statistical data is created and sent to a database for later use by the researchers. All other connected clients - Spiritus Ludi Applications will receive the digital high-five and forward the data to the connected sleeve where it will execute the appropriate logic. It is likely there will be multiple digital high-fives incoming from different users in the team in a short period of time. The sleeve will queue these digital high-fives and play them to the user by running the haptic motor in the sleeve repeatedly for a set period until the queue reaches zero.

Figure 1. A high-level overview of how the Spiritus Ludi System works from when a user interacts with the touch sensor to send a digital high-five. (Jumisko-Pyykkö et al., 2023)



The Spiritus Ludi Smart Sleeves are designed to allow team communication through mediated touch. By doing so, it aims to encourage a team's positive emotional state and through this, improve their overall performance. The design process and the accompanying results can be found in Spiritus Ludi: Smart Wearable Textiles that Foster and Elevate Connections in E-sports Teams (Jumisko-Pyykkö et al., 2023).

2.3 Software Design

The two primary systems used for data transfer in Spiritus Ludi are SignalR and Bluetooth Low Energy (BLE). These are well developed and widely used systems. The characteristics of these systems are ideal for our use case.

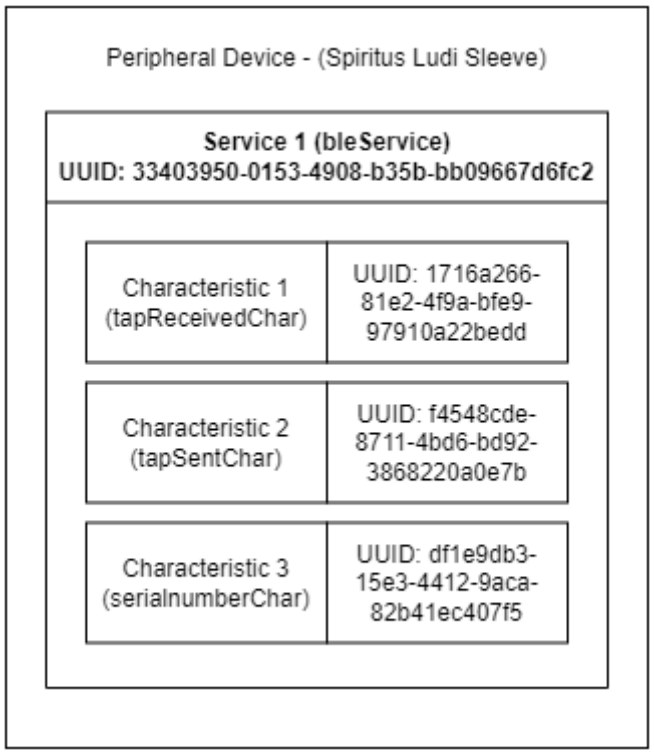
2.3.1 Bluetooth Low Energy

Bluetooth Low Energy at the physical layer is a radio transmitter/receiver which operates in the 2.4GHz band. After this there are several layers which create the BLE core specification. Delving into the working parts of BLE theory is outside the scope of this thesis. It can become a very

complicated subject, however documentation has been created to explain the inner workings of BLE such as The Bluetooth® Low Energy Primer (Woolley & Anees, 2024)

The following will cover the theory needed for this implementation. This BLE implementation works on a one-to-one connection basis. One device is the Central Device, and the other device is known as the Peripheral Device. Between these two devices, a persistent connection can be made for data transfer. The Peripheral Device will have Services which it can advertise. These Services contain Characteristics which can hold pieces of data a Central Device would be interested in. For a Central Device to receive this data, it will connect to the Peripheral Device and the subscribe to the Characteristics that have the properties of `BLERead` and `BLENotify`. `BLENotify` allows the Peripheral to let the Central Device know when a value has changed, the Central can then read the new value through the `BLERead` property. This prevents the Central having to constantly monitor the characteristic for changes on value, thus saving resources. The Peripheral Device will receive data from the Central Device using the characteristics with the property of `BLEWrite`, denoting to the central device that the data assigned to that characteristic is writable. The Central Device will have received and stored the characteristic information so it will know how to appropriately handle the sending and receiving of data.

Figure 2. A diagram for the BLE Service and Characteristics of the Spiritus Ludi Sleeves with example Universally Unique Identifiers (UUIDs).



The Universally Unique Identifiers (UUIDs) are used by the Central Device to identify the characteristics and their properties. This implementation as seen in Figure 2 is known as the Generic Attribute Profile (GATT). This GATT profile defines how data is organized and back and forth via the Services and Characteristics. (Bluetooth Low Energy Services, a Beginner's Tutorial - Bluetooth Low Energy - nRF5 SDK Guides - Nordic DevZone, 2015)

2.3.2 SignalR

Microsoft defines SignalR as an "open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly." (Overview of ASP.NET Core SignalR | Microsoft Learn, n.d.). SignalR allows for Remote Procedure Calls (RPC) (Remote Procedure Call - IBM Documentation, n.d.) which allows for functions on the client side of communication be invoked from the server side. In this case, it is a .NET application hosting the SignalR Hub for Spiritus Ludi. The SignalR Hubs are high level pipelines that allow clients and servers to call methods on each other using RPC (Use Hubs in ASP.NET Core SignalR | Microsoft Learn, n.d.).

SignalR uses three modes of transport for handling real time communication. They are WebSockets, Server-Sent Events, and Long Polling. Out of these three, WebSockets are the only protocol which provide a true persistent bi-directional connection. Long Polling connections will stay open, however will either time out after a specified time or the client will be required to make a new connection request immediately after data has been received from the server. SSEs allow servers to push real time updates to clients over a single HTTP connection, this making it unidirectional.

The SignalR library will choose the most appropriate method of transport, based on the capabilities of the server and client. SignalR can be forced into a method of transport if necessary and it provides a graceful fall back between each of the three transport methods, therefore not requiring any manual management in development. For Spiritus Ludi, the SignalR Library manages which connection mode to use. The optimal connection method for the Spiritus Ludi systems is WebSocket.

3 Methods and Techniques used to Develop Spiritus Ludi's Social Mediated Touch System

Spiritus Ludi was designed, developed and field tested in conjunction with over 100 professional esports players and coaches, developed under the Wearable Intelligence Competence Development Project and Pirkanmaa's Knowledge and Innovation Ecosystem Project for Sustainable and Smart Textiles Project and funded by the European Union Regional Development Fund.

The software aspect of Spiritus Ludi was only one part of many in the platform's development. Task and project management for the software development was integrated into the overarching project management. Overall project management was handled by the Principal Research Scientist and the Development Specialist. The software development supervision was handled by the department's senior developer. This was done with bi-monthly meetings, where guidance was available.

Kanban boards were created to track task progress, along with delays and problems. A variation of the agile scrum approach was adopted, having weekly meetings to brief on progress, adjusting courses of action and direction as necessary to manage goals and tasks. This allowed the team to organise necessary collaboration and work on the larger common objectives.

Code was stored in GitHub, using version control and branching to develop and integrate features. For ease of code base management, three internal Git repositories were created in GitHub. One for the Spiritus Ludi React Native Phone Application, one for the .NET Application running SignalR, and one for the Spiritus Ludi Sleeve Firmware code. Later in the development, a staging branch was added, to test new feature branches with the base of the current main branch, allowing for the main branch to be used for the deployed phone application.

The React Native application distribution for field testing was achieved using the Google Play Console platform. This platform allows for deployed applications to be downloaded via the Google Play Store. Our department as an account and it includes an "Internal Testing" feature that allows for deployed applications to be downloaded only to certain listed accounts. This was suitable for our field testing needs at this stage.

4 The Aim and Purpose of the Spiritus Ludi Software

The aim of the software in Spiritus Ludi is to allow low latency communication between the users in a team environment using touch, while also recording user interaction without impacting on the communication latency. Conventional touch between humans is often registered within milliseconds. To emulate the speed of touch communication on contact, the communication systems implemented needed to have similar characteristics, coupled with a structure of how to use the different types of digital communication available.

4.1 Required Software Characteristics

For Social Mediated Touch to be viable in Spiritus Ludi, the software needed to have the following characteristics to ensure the development of a successful prototype. These characteristics are defined as reliability, low latency, and sustainability. Not implementing these characteristics would have a negative impact on the performance of Spiritus Ludi.

Reliability is necessary for such a system as it needed to mimic human touch. The digital high fives transmitted and received need to be executed in a consistently reliable manner. If team members cannot trust that the sleeve will send or receive digital high-fives, this will influence how, or even if they use it.

In normal circumstances, the act of a high-five will provide instantaneous feedback between the participants. In a digital context this would mean zero latency. However, latency between computers over various forms of communication can occur due to factors outside the control of the user or the developer. Examples of this include a low internet connection speed, an internet service outage, processing run time being taken by external programs and processes, and hardware limitations. Therefore, the data sent and received needs to be minimal to ensure the fastest possible data transfer.

Sustainability in software covers a multitude of aspects. In the context of Spiritus Ludi, it refers to the resources needed to keep the developed platform functional and secure. Spiritus Ludi needs to maintain a useable state after active project development had ceased. Using various Azure Services which are readily available to the department, and popular frameworks which are well documented and maintained, has allowed for the creation of such a platform.

4.2 A High-Level Software Overview of Spiritus Ludi

Spiritus Ludi software is divided down into three sections. The firmware on the sleeve, the phone application which allows users to connect their Spiritus Ludi sleeve to their phone, and Azure Cloud hosting the .NET application running the real-time communication SignalR Instance.

Figure 3. A high-level end-to-end diagram of how data is transmitted and received, and the technologies used to achieve this.

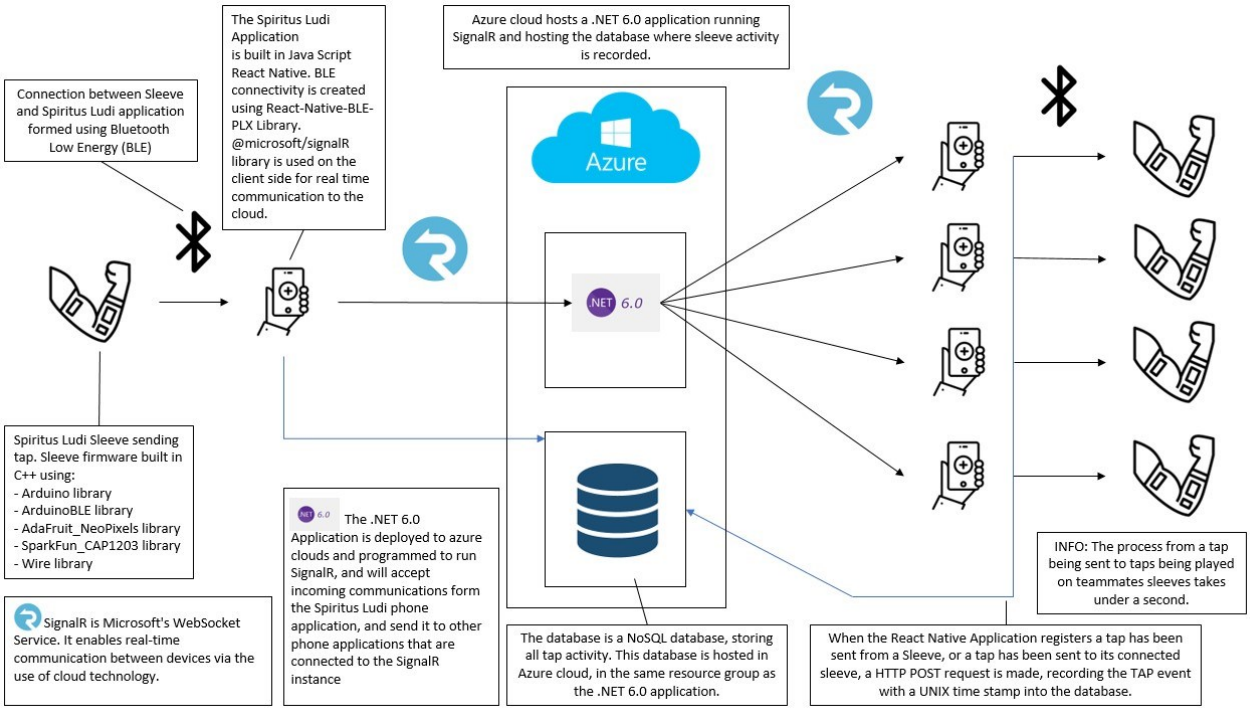


Figure 3 illustrates the how each of the communication methods are used to create end-to-end communication between all sleeves. The bidirectional communication allows all sleeves to send and receive digital high-fives as long as they are connected to the appropriate services.

4.2.1 The Spiritus Ludi Sleeve

The Spiritus Ludi Sleeve uses a Seeed Studio XIAO nRF52840, a small form factor board with a single thread ARM processor. Using libraries available through the Arduino IDE, a program was created using non-blocking methods to quickly register touch input from the user and then transmit that via the onboard Bluetooth Low Energy (BLE) chip to the Spiritus Ludi phone application.

The strength of BLE is its ability to send small amounts of data at speed with minimal energy consumption compared to its predecessors, which is suitable for our use case. Users send and

receive multiple digital high-fives in short periods of time, each interaction creating a small amount of data to be transmitted. The Arduino development environment uses C++ to create firmware that can be uploaded to boards such as the Seeed Studio XIAO nRF52840.

4.2.2 The Spiritus Ludi Phone Application

This application is a React Native phone application. React Native was chosen for its ability to run on Android and iOS and its well documented and actively developed framework. For this project, the application was developed for Android handsets only. The handsets used for development and field testing were Nokia G11 running the Android 13 operating system. This React Native Application enables bi-directional communication to and from the Spiritus Ludi sleeves worn by players in a team. This was achieved using the React-Native-BLE-PLX and the @microsoft/signalr client Node Package Manager (NPM) packages.

4.2.3 The Spiritus Ludi Real Time Communication

For the real time communication, a .NET 6 application was deployed to the project Azure Resource. This application created a SignalR Service, a “hub” which allowed clients with the correct connection string to establish a connection with the SignalR Service. SignalR is suitable as it allowed us to push data from a server to a client in real time. SignalR utilizes WebSocket technology as its preferred mode of transport, but it also allows for Server-Sent Events and Long Polling as contingency when WebSocket transport is not available. This prevents a single point of failure by not relying on just one transport protocol.

In parallel to the SignalR connection asynchronously, data is also sent to a NoSQL database hosted in the same Azure resource. This records user activity, which was used in field testing and research. The user data was sent in a parallel instead of with the digital high-five data via SignalR to ensure that processing time when sending user interaction data was kept to a minimum. User interaction data was sent via a HTTP POST request to an azure storage endpoint hosting the NoSQL database.

5 The Practical Implementation of Spiritus Ludi

As mentioned previously, low latency, reliability, and sustainability were the core characteristics needed in the platform's software. As the Spiritus Ludi platform is made up of three parts mentioned in section 4.2, the development followed a similar structure.

5.1 The Spiritus Ludi Sleeve Firmware

The Spiritus Ludi sleeves are a prototype system. For this reason, the Arduino development environment was chosen to create the program to run on the Seeed Studio XIAO nRF52840 microcontroller. The environment was created by following the documentation provided by Seeed Studio (Getting Started with XIAO nRF52840 | Seeed Studio Wiki, n.d.). Table 1 provides the information for the required libraries in the firmware.

Table 1. A table listing the libraries used in the sleeve firmware, the respective version, and the purpose they serve.

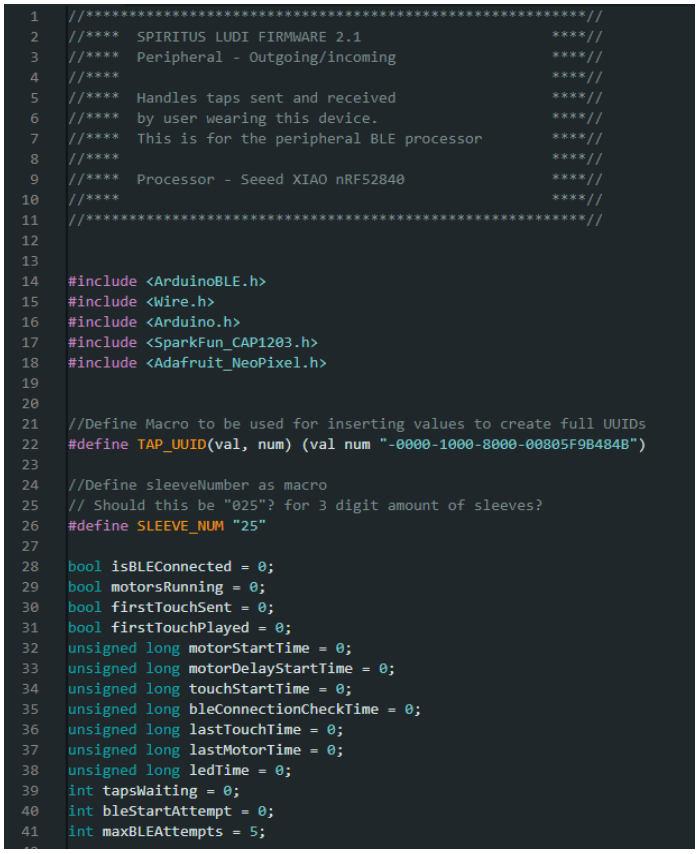
Library	Version	Purpose
Arduino IDE (bundles with Wire.h and Arduino.h)	2.3.3	Arduino.h required for specific chip support, in this case to use the nRF52840 using the appropriate board support package. Wire.h required to communicate with connected I2C devices.
ArduinoBLE.h	1.3.7	Enables the Arduino program to interface with the Bluetooth module.
SparkFun_CAP1203.h	1.0.5	Required for interaction with the I2C based capacitive sensor.
Adafruit_Neopixel.h	1.12.3	Required for controlling single-wire-based LED pixels used for the sleeve UI.

The file that contains the program is called SL_Firmware.ino. The current firmware version installed on the Spiritus Ludi Sleeves at the time of writing this is 2.1. The BLE communication here is set up as the peripheral device.

5.1.1 The Firmware Setup

Arduino scripts run with imports, definitions and global variables typically listed at the top of the file. Following this is then the `setup()` function which runs only once on every power up or restart and is used to initialize variables, libraries and other requirements the program may need before the next phase. Then the `loop()` function which runs indefinitely executing code placed inside it. A flow chart of this one-time process is shown in Figure 11.

Figure 4. An image of the Spiritus Ludi Sleeve Firmware code showing the imported libraries and global variables created.



```
1  //*********************************************************//
2  //****  SPIRITUS LUDI FIRMWARE 2.1                     ****//
3  //****  Peripheral - Outgoing/incoming                 ****//
4  //****                                                 ****//
5  //****  Handles taps sent and received                 ****//
6  //****  by user wearing this device.                   ****//
7  //****  This is for the peripheral BLE processor       ****//
8  //****                                                 ****//
9  //****  Processor - Seeed XIAO nRF52840                ****//
10 //****                                                 ****//
11 //*********************************************************//
12
13
14 #include <ArduinoBLE.h>
15 #include <Wire.h>
16 #include <Arduino.h>
17 #include <SparkFun_CAP1203.h>
18 #include <Adafruit_NeoPixel.h>
19
20
21 //Define Macro to be used for inserting values to create full UUIDs
22 #define TAP_UUID(val, num) (val num "-0000-1000-8000-00805F9B484B")
23
24 //Define sleeveNumber as macro
25 // Should this be "025"? for 3 digit amount of sleeves?
26 #define SLEEVE_NUM "25"
27
28 bool isBLEConnected = 0;
29 bool motorsRunning = 0;
30 bool firstTouchSent = 0;
31 bool firstTouchPlayed = 0;
32 unsigned long motorStartTime = 0;
33 unsigned long motorDelayStartTime = 0;
34 unsigned long touchStartTime = 0;
35 unsigned long bleConnectionCheckTime = 0;
36 unsigned long lastTouchTime = 0;
37 unsigned long lastMotorTime = 0;
38 unsigned long ledTime = 0;
39 int tapsWaiting = 0;
40 int bleStartAttempt = 0;
41 int maxBLEAttempts = 5;
```

In Figure 4, lines 28 to 41 create the different types of variables needed to manage the digital high-fives input by the user and received from the React Native Application.

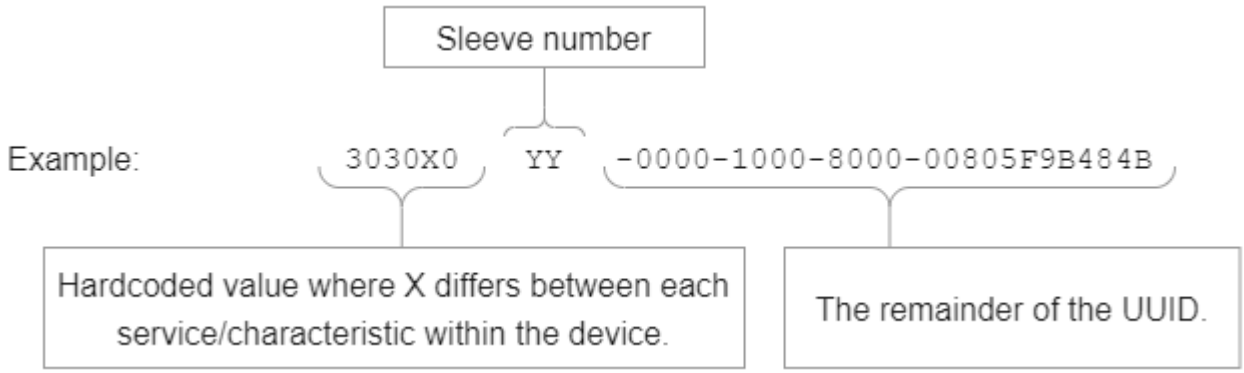
In Figure 5, line 54 creates part of the Local Device name “Spiritus Ludi ” in an array. Lines 57 through to 62 assign the UUIDs to the BLE Service and Characteristics.

Figure 5. UUIDs being assigned to the sleeves BLE Service and Characteristics. These will later be advertised and available for connection.

```
43  // Initialize the CAP1023 chip instance
44  CAP1203 Touch;
45
46  // Getting serialnumber from the physical device
47  uint64_t serialNumber = (uint64_t(NRF_FICR->DEVICEADDR[1]) << 32) | NRF_FICR->DEVICEADDR[0];
48  // Don't remove, These are made for BLE Characteristic. uint64_t too long long for Arduino String() function
49  uint32_t serialNumberPartOne = NRF_FICR->DEVICEADDR[1];
50  uint32_t serialNumberPartTwo = NRF_FICR->DEVICEADDR[0];
51
52  //Create value to be used for the local name.
53  //See Documentation for naming convention
54  char sleeveName[] = "Spiritus Ludi ";
55
56  //Assigns bluetooth service to UUID of XXXXXXXX-0000-1000-8000-00805F9B484B and named bleService
57  BLEService bleService(TAP_UUID("303010", SLEEVE_NUM));
58
59  //Create Characteristics. BLERead allows Characteristic to be read. BLENotify notifies central value has changed.
60  BLEByteCharacteristic tapReceivedChar(TAP_UUID("303020", SLEEVE_NUM), BLEWrite);          //** Give write permissions **// From App to Sleeve
61  BLEByteCharacteristic tapSentChar(TAP_UUID("303030", SLEEVE_NUM), BLERead | BLENotify);   // From Sleeve to App
62  BLEStringCharacteristic serialnumberChar(TAP_UUID("303040", SLEEVE_NUM), BLERead, 16);    // Initializing BLECharacteristic for serialnumber
63
```

This program creates the Service and Characteristic UUIDs using 3 pieces of data. The first part being hardcoded data `3030X0`, “X” being replaced with either 1, 2, 3, or 4 as seen on lines 57 and 60 to 62. The second part are two characters which are defined by the sleeve number macro on line 26 identified as `SLEEVE_NUM`. This is appended to the end of the first part. The third part is found on line 22 – seen in Figure 4. `TAP UUID` accepts these two values as arguments `val` and `num`. They are prepended to the hardcoded value `-0000-1000-8000-00805F9B484B`. `TAP_UUID` is then used for each UUID.

Figure 6. A breakdown of the 128-bit BLE UUIDs used by the Spiritus Ludi Sleeves.



The variation between each UUID is then contained in the X value, and the sleeve number part of the UUID, denoted as YY ensures that UUIDs are not identical between devices as seen in Figure 6.

Figure 7. The variables on lines 97 and 98 are used as a value of time in milliseconds. These values need to be sufficiently long enough for the motor to register the signal to turn on and off.

```
97  int MotorRunTime = 200;   // Number of milliseconds the motor needs running
98  int MotorRestTime = 160;  // Time between motor plays **ORIGINAL TIME 160MS
99
```

At line 106, the `setup()` function is executed. The `SLEEVE_NUM` variable is concatenated to the `sleeveName` variable on line 110. This creates the human readable ID that will be shown on the React Native application as discovered Spiritus Ludi sleeves.

During the `setup()` execution, various checks are conducted to ensure the hardware is working as expected. Errors/Faults will be displayed to the user via the sleeve UI - Figure 8.

Figure 8. The BLE module initialisation is completed in a while loop. If the BLE module does not successfully initialise within 5 attempts – set on line 41, then the program will enter an eternal while loop on line 175 to prevent it from progressing to the `loop()` function.

```
162  //Begin BLE initialisation
163  while (bleStartAttempt < maxBLEAttempts) {
164    if (!BLE.begin()) {
165      Serial.println("BLE.begin: FAILED. Attempt " + String(bleStartAttempt + 1) + " of " + String(maxBLEAttempts) + ".");
166      bleStartAttempt++;
167      BLE.end();  // Terminates Bluetooth module for a clean restart.
168      delay(500);
169      if (bleStartAttempt >= maxBLEAttempts) {
170        BTStatusR = Intense;
171        InfoPad(PwrStatusR, PwrStatusG, BTStatusG, BTStatusB, BTStatusR, BtyStatusR, BtyStatusG, MessageG, MessageB);
172        delay(100);
173
174        Serial.println("BLE module failed to start");
175        while (1)
176          ;  // Holds start up on infinite loop as BLE cannot be initialised after 5 attempts
177      }
178    } else {
179      Serial.println("BLE module: OK.");
180
181      //InfoPad: BT connect succesfully
182      BTStatusG = Intense;
183      InfoPad(PwrStatusR, PwrStatusG, BTStatusG, BTStatusB, BTStatusR, BtyStatusR, BtyStatusG, MessageG, MessageB);
184      delay(1000);
185      BTStatusG = 0;
186      InfoPad(PwrStatusR, PwrStatusG, BTStatusG, BTStatusB, BTStatusR, BtyStatusR, BtyStatusG, MessageG, MessageB);
187      delay(500);
188      break;
189    }
190  }
```

On successful initialisation, the while loop enters in the else statement on line 178, The BLE LED on the sleeve UI will flash green once, and then `break;` is used to exit the loop. If the BLE module fails to initialise within the number of permitted attempts defined by the variable `maxBLEAttempts`, next the BLE LED will turn a solid red colour denoting to the user the sleeve

should be restarted. If the BLE module fails to start after three device restarts, then it is likely there is faulty hardware that will require replacing.

On line 197 - Figure 9, the human readable device name is set. The value defined by the variable `sleeveName` is used in the `BLE.setLocalName()` argument and will be advertised to central devices. Line 207 to sets the bleService UUID as a discoverable advertised service. Lines 209 to 211 then adds the Characteristics to the bleService. The newly created service and the respective characteristics are added to the BLE instance on line 214. Now when the service UUID is queried for usable characteristics, the UUIDs added on lines 209 to 211 will be returned.

Figure 9. The properties required to create the BLE service and associated characteristics in the `setup()` function.

```
196   //Set local name that will be seen on the app. It is the name shown on the central is connect to.
197   BLE.setLocalName(sleeveName);
198   //Prints local name
199   Serial.print("THIS DEVICE: ");
200   Serial.println(sleeveName);
201
202   // Print device SerialNumber
203   Serial.print("Device SerialNumber: ");
204   Serial.println(serialNumber, HEX);
205
206   //Set the UUID for the service this peripheral will advertise.
207   BLE.setAdvertisedService(bleService);
208   //Add characteristics to bleService
209   bleService.addCharacteristic(tapReceivedChar);
210   bleService.addCharacteristic(tapSentChar);
211   bleService.addCharacteristic(serialnumberChar);
212
213   //Add service (bleService) which have attached characteristics
214   BLE.addService(bleService);
```

The `tapSentChar` and `tapReceivedChar` characteristic values are then set to zero to ensure no other value is written on start up into these characteristics. Line 224 is different as `serialnumberChar` will only ever hold the `serialnumberFull` value. This value will not be changed after startup.

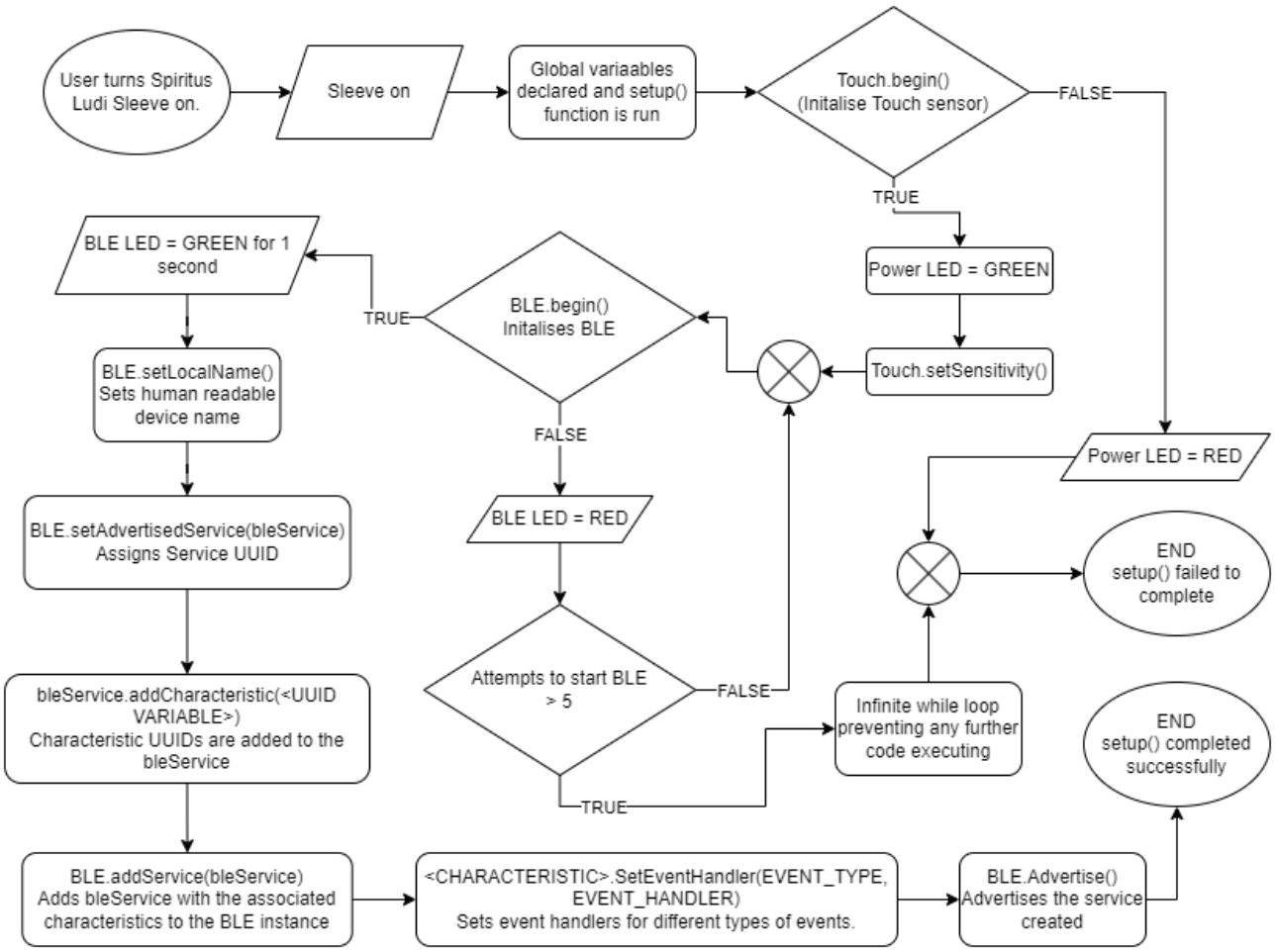
Event Handlers are then set using the BLE setEventHandler() callback function. The first argument is what type of event occurred and the second argument is the function that is to be executed in the event of the type of event occurring.

Figure 10. An image of code showing event handlers being assigned to the different BLE characteristics.

```
226   //Assign event handlers for connected/disconnected to peripheral. Consider using
227   BLE.setEventHandler(BLEConnected, connectHandler);
228   BLE.setEventHandler(BLEDisconnected, disconnectHandler);
229
230   //Assign event handlers for each characteristic. Updated, subscribed, and unsubscrbed event handlers for each characteristic.
231   //Functions eg: characteristicUpdated, characteristicSubscribed, characteristicUnsubscribed are found after the loop function.
232   tapSentChar.setEventHandler(BLEUpdated, characteristicUpdated);  //Do we need this?
233   tapSentChar.setEventHandler(BLESubscribed, characteristicSubscribed);
234   tapSentChar.setEventHandler(BLEUnsubscribed, characteristicUnsubscribed);
235
236   tapReceivedChar.setEventHandler(BLEWritten, addTap);
237   tapReceivedChar.setEventHandler(BLESubscribed, characteristicSubscribed);
238   tapReceivedChar.setEventHandler(BLEUnsubscribed, characteristicUnsubscribed);
239
240   //Start Adveristing. BLE will now advertise to central devices with the information above
241   BLE.advertise();
242   Serial.println("Bluetooth ok. Awaiting conn");
243 }
```

The event types in the first argument position accepts BLE device events and characteristic events, such as the device events BLEConnected and BLEDisconnected on lines 227 and 228 respectively. On line 241 the last BLE function is to set the BLE instance into an advertising mode before entering the loop function. From this point the Spiritus Ludi Sleeves will be discoverable and connectable while in an advertising state.

Figure 11. A flow chart of the `setup()` function. Upon successful completion, the `loop()` function will be indefinitely run until the sleeve is switched off.

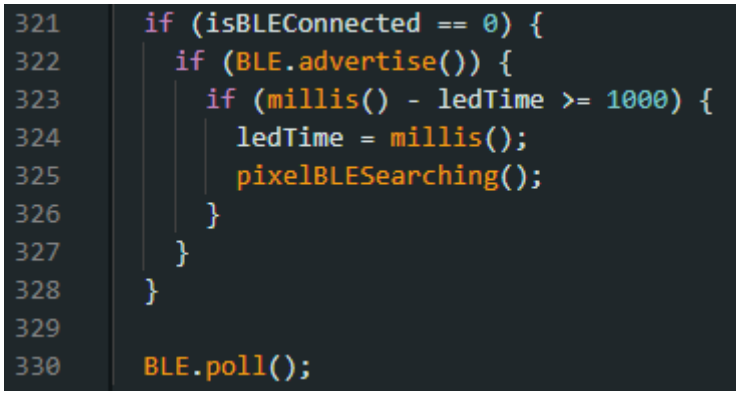


The `setup()` function gives the ideal opportunity to ensure everything is working as expected before the user can start using the sleeve as intended.

5.1.2 The Loop Function

The loop function is an infinite loop that will reiterate continuously after the successful completion of the setup function. This design is default for Arduino sketches. This loop iterates at a fast rate, but the results of this iteration speed will be determined by various factors like the processing capability of the processor used, and the complexity of the code inside the loop.

Figure 12. This if statement will execute if the sleeve is in a disconnected state. It will make itself discoverable and control the BLE UI to show an advertising state.



```
321   if (isBLEConnected == 0) {
322     if (BLE.advertise()) {
323       if (millis() - ledTime >= 1000) {
324         ledTime = millis();
325         pixelBLESearching();
326       }
327     }
328   }
329
330   BLE.poll();
```

In the `loop()` function, due to the constraint of a single thread processor, the use of non-blocking functions is strongly recommended. Many of the functions required for the sleeve to operate as expected are based on the passing of time. However, the use of the `delay()` function is avoided where possible. To tackle this, the use of `millis()` is employed to track the passing of time without blocking the program. The `millis()` function is built in into the Arduino base library and returns the number of milliseconds that have passed since the current program has started running. This is value is returned as an `unsigned long`. By using this method, the time stamp is stored in an `unsigned long` variable, and an `if` statement is used to check the difference in time passed since the last iteration of the `loop()` function. Figure 12 shows on line 323 if the latest stored `ledTime` subtracted from the current `millis()` returned value. Is equal to or greater than `1000` milliseconds. If this statement returns `true`, then `ledTime` is updated with the new current `millis()` returned value and the `pixelBLESearching()` function is executed. If this statement returns `false`, the updating the `ledTime` value and the `pixelBLESearching()` function will be skipped. This method allows for the tracking of time dependent events with suitable accuracy while being able to handle other events and user input during the time tracked period. For the Spiritus Ludi sleeves, the `loop()` function which starts on line 246 is primarily split into two `if()` statements, found on lines 248 - Figure 13 and 321 - Figure 12. These check the BLE connection state. After the initial set up, the sleeve will enter the `if` statement on line 321. This `if` statement is entered when the BLE status Boolean `isBLEConnected` is set to `false`. As seen in Figure 4 on line 28 this Boolean is initially set to `0` (false). This will be updated with the event handlers depicted in

Figure 10.

Upon successful connection with a central device – in this case the Nokia G10 handset running the Spiritus Ludi React Native Application, the loop enters and repeats the if statement on line 248. This checks that the device is still connected. Lines 250 to 260 handles when the touch sensor has been touched by the user.

Figure 13. Provided the sleeve is connected to a central device, the code here will execute when the user interacts with the touch sensor.

```
246  void loop() {
247
248    if (isBLEConnected == 1) {
249      //if 1203 touched, tap sent.
250      if (Touch.isTouched() == 1) {
251        if (!firstTouchSent) {
252          pixelsSendTapOn();
253          firstTouchSent = 1;
254        }
255        tapSentChar.setValue(1);
256        lastTouchTime = millis();
257        Serial.println("Touch sent");
258        while (Touch.isTouched() == 1)
259          ;  // Stops streaming of touch recording
260      }
261      if (millis() - lastTouchTime > 500) {
262        msgPixelsOff();
263        firstTouchSent = 0;
264      }
```

The `firstTouchSent` variable is a Boolean that sets the state if the first touch registered by the touch sensor has been sent. If `Touch.isTouched()` is true, but `firstTouchSent` is `false`, then this means there are unsent high-fives from this sleeve. This triggers the function `pixelsSendTapOn()` to set the appropriate LED UI and sets `firstTouchSent` to `true`. If both `Touch.isTouched()` and `firstTouchSent` are `true`, then line 255 where the value of the `tapSentChar` is set to `1`. `lastTouchTime` is then sent set to the current `millis()` value and is repeatedly set with every new registered touch. The while loop on line 258 will prevent the streaming of touches. The streaming means multiple touches would be registered every second. The while loop has no executable code inside it, however while the touch sensor is registered as being touched, then it will not register a new touch until it is released. The `lastTouchSent` variable is used to check how much time has passed since the last registered touch. If 500 milliseconds have passed since the last registered touch, the `msgPixerlsOff()` function is executed to turn off the message UI LEDs that tell the user when a digital high-five is being sent of received.

Figure 14. The process handling if there is exactly one digital high-five waiting to be played.

```
267     if (tapsWaiting == 1) {
268       firstTouchPlayed = 1;
269       if (firstTouchPlayed == 1 && firstTouchSent == 0) {
270         pixelReceiveTapOn();
271       }
272       if (!motorsRunning && (millis() - motorDelayStartTime >= MotorRestTime)) {
273         motorsRunning = 1;
274         digitalWrite(MotorP, HIGH);  // Switch the motor ON
275         motorStartTime = millis();
276       }
277       if (motorsRunning && (millis() - motorStartTime >= MotorRunTime)) {
278         tapsWaiting--;
279         motorsRunning = 0;
280 
281         digitalWrite(MotorP, LOW);
282         motorDelayStartTime = millis();
283       }
284       if (tapsWaiting == 0 && (millis() - motorDelayStartTime >= 2000) && firstTouchPlayed == 1) {
285         firstTouchPlayed = 0;
286         msgPixelsOff();
287       }
288     }
```

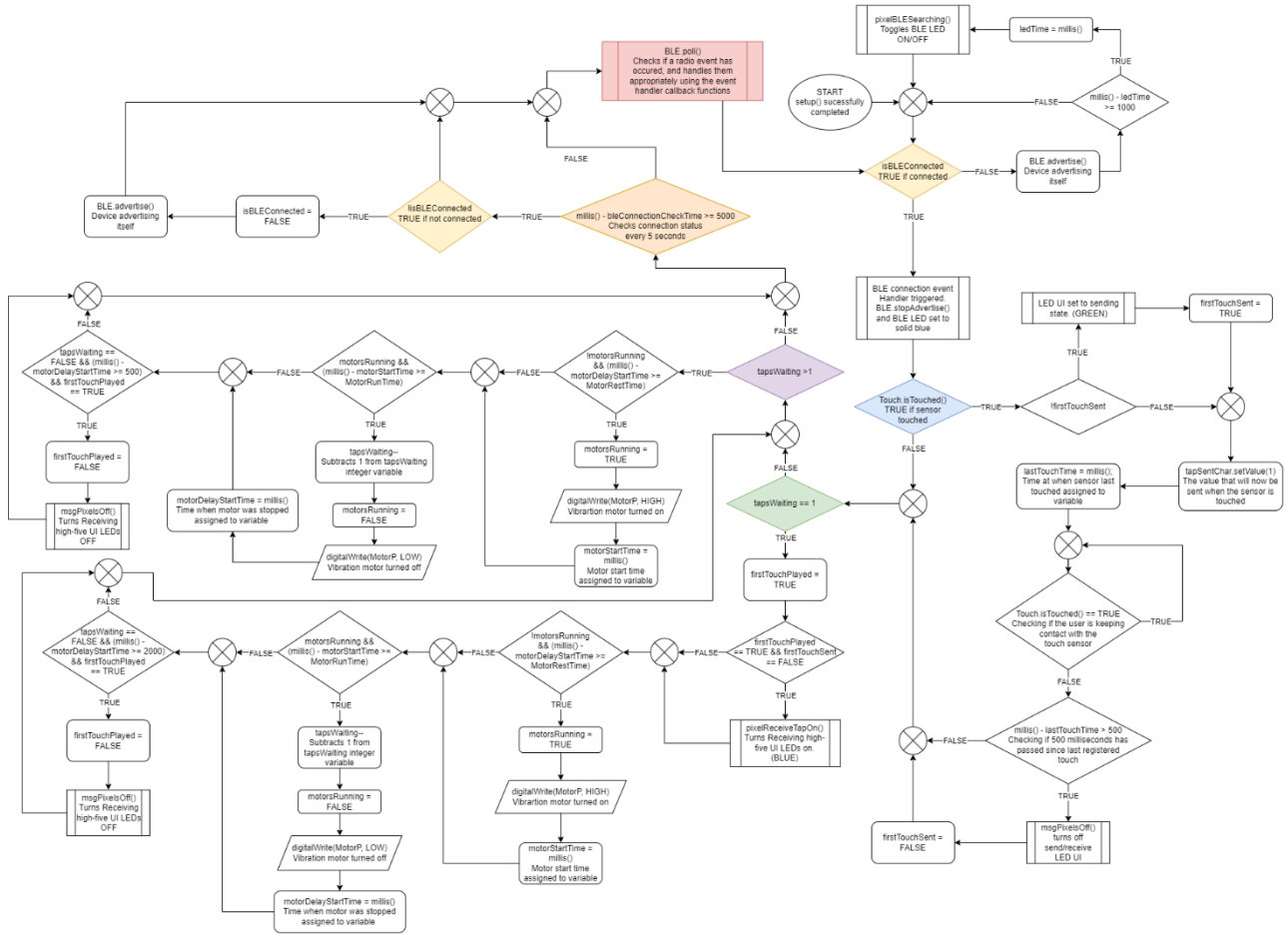
The code from lines 267 to 288 seen in Figure 14 shows the logic if there is only one digital high-five waiting to be played – denoted by the variable `tapsWaiting`. This block controls when the haptic motor and UI LEDs related to receiving digital high-fives will start and stop. The period of time the motor is run and the period of time for breaks in the motor running is defined on lines 97 and 98 - Figure 7. All using the `millis()` function to measure the passing of time. Figure 15 is executed when there is more than one digital high-five waiting to be played.

Figure 15. The process handling if there is more than one digital high-five waiting to be played.

```
290     if (tapsWaiting > 1) {
291       if (!motorsRunning && (millis() - motorDelayStartTime >= MotorRestTime)) {
292         motorsRunning = 1;
293         digitalWrite(MotorP, HIGH);  // Switch the motor ON
294         motorStartTime = millis();
295         //Serial.println("Motor On");
296       }
297       if (motorsRunning && (millis() - motorStartTime >= MotorRunTime)) {
298         tapsWaiting--;
299         motorsRunning = 0;
300 
301         digitalWrite(MotorP, LOW);
302         motorDelayStartTime = millis();
303         //Serial.println("Motor Off");
304       }
305       if (tapsWaiting == 0 && (millis() - motorDelayStartTime >= 500) && firstTouchPlayed == 1) {
306         firstTouchPlayed = 0;
307         msgPixelsOff();
308       }
309     }
```

The following flow chart - Figure 16 provides a flow chart of the `loop()` function and how it handles events on the single thread. Yellow highlights the BLE Connection handling, Blue denotes the process if the touch sensor has been touched, green denotes if there is exactly one digital high-five waiting to be played, purple denotes if there is more than one digital high-five waiting to be played, orange denotes where the BLE Connection state is checked, and Red denotes where the BLE Connection is polled for any changes in data received.

Figure 16. A flow chart of the `loop()` function inside the Spiritus Ludi firmware



5.2 The Spiritus Ludi Phone Application

React Native was the chosen framework for the phone application. When the phone application was initialized, React Native version 0.71 (Introduction · React Native, n.d.) was the version in active development. However, developing the Spiritus Ludi Phone Application is possible in later versions.

The code may make references to “TapApp”. This was the name given before the name Spiritus Ludi was given.

Table 2. A table listing the relevant dependencies used to enable real time communication between the connected sleeves over the internet.

Library	Version	Purpose
@microsoft/signalr	7.0.2	The client library to enable persistent communication with the SignalR Hub and handle

		incoming and outgoing events. (@microsoft/Signalr - Npm, n.d.)
react-native-ble-plx	2.0.3	The library used to create a Central BLE instance to communicate peripheral devices. (React-Native-Ble-Plx - Npm, n.d.)
react-native-device-info	10.3.0	A library used to call for information about the physical device and the associated operating system – in this case the Nokia G10 and the Android OS. (React-Native-Device-Info - Npm, n.d.)
react-native-permissions	3.6.1	A library used to request and check permissions granted or rejected from the Android OS. In this case, Bluetooth permissions. (React-Native-Permissions - Npm, n.d.)
react-native-quick-base64	2.0.5	A library to enable encoding data from binary to Base64 string and decode Base64 back to binary. (React-Native-Quick-Base64 - Npm, n.d.)
react-redux	8.0.5	A library used to store state and actions to React components, simplifying the storing and calling of state logic in applications.(React-Redux - Npm, n.d.)

At the time of writing, these dependencies - Table 2, have since been replaced with newer versions. As per best practices, it is encouraged to use the latest stable versions at the time of development.

The react native application was created using the React Native CLI Environment set up. The environment creation is well documented. When the creator of this thesis joined the Future Wearables teams it was asked if the use of push notifications could be used. This was briefly tested using Firebase Cloud Messaging; however, it was found that push notifications, at least in the development environment took upwards of twenty seconds. It was deemed unsuitable, and no further time or resources were spent on looking into the use of push notifications.

5.2.1 Using Bluetooth Low Energy in the React Native Application

Bluetooth connectivity has certain requirements for appropriate connectivity and functionality to be possible with the Spiritus Ludi Sleeve. Creating the connection, scanning for peripherals, connecting to the device, reading the available services and characteristics, listening for data from the sleeve, sending data to the sleeve and disconnecting from the sleeve. This instance and functionality have been created in a singleton class named `BleTest`.

5.2.2 Creating a Connection

Firstly, before the BLE-PLX package can be utilised, permissions need to be obtained from the user - Figure 17. As per the Android Operating System documentation regarding Bluetooth connectivity, `BLUETOOTH_SCAN` and `BLUETOOTH_CONNECT` are required. The Spiritus Ludi phone application also `ACCESS_FINE_LOCATION`, although it is never used. When using BLE, it is assumed that using the Bluetooth scan results will be for attaining physical location, unless it is strongly asserted that physical location will never be used. This can be done by using the `android:usesPermissionsFlags` attribute with the `BLUETOOTH_SCAN` permission and setting the attribute's value to `neverForLocation`. (Bluetooth Permissions | Connectivity | Android Developers, n.d.). During development, this caused the application to crash. Time was spent trying to resolve this, however, due to the limited time available and knowing the sleeves were going into anonymous field-testing conditions, it was deemed that allowing location permissions would be acceptable in this instance, although not desirable.

Figure 17. An image of the asynchronous `resquestPermissions()` function in file bleTest.js used to obtain permissions to access BLE functions on the operating system level, then checks that permission has been obtained.

```
52
53  static async requestPermissions() {
54    if (Platform.OS === 'android') {
55      const result = await requestMultiple([
56        PERMISSIONS.ANDROID.BLUETOOTH_SCAN,
57        PERMISSIONS.ANDROID.BLUETOOTH_CONNECT,
58        PERMISSIONS.ANDROID.ACCESS_FINE_LOCATION,
59      ]);
60      // Checks permissions status
61      const isGranted =
62        result['android.permission.BLUETOOTH_CONNECT'] === PermissionsAndroid.RESULTS.GRANTED &&
63        result['android.permission.BLUETOOTH_SCAN'] === PermissionsAndroid.RESULTS.GRANTED &&
64        result['android.permission.ACCESS_FINE_LOCATION'] === PermissionsAndroid.RESULTS.GRANTED;
65      if (isGranted === true) {
66        console.log('Android Bluetooth Permissions Granted');
67      }
68    } else {
69      console.log('Error: Currently only available on Android');
70    }
71  }
72
```

5.2.3 Scanning for Peripherals

The BLE communication in the Spiritus Ludi phone application is set up as the central device. The scanning function - shown in Figure 18, has been implemented to scan for peripherals for ten seconds using a `Promise()`. After 10 seconds of scanning, the `Promise()` with either `resolve()` with the `setTimeout()` triggering the `stopDeviceScan()` function, or the `promise()` will `reject()` with warning errors.

Once the 10 second timer has started, the application will scan for devices, the nested if statements check that the device scanned does not have a name and device ID value of `null`, then checks that the name of the peripheral device contains “Spiritus Ludi” - line 82. In Android, the device ID will be the peripheral Media Access Control (MAC) address. In iOS, it will be the peripheral Universally Unique Identifier (UUID)

Figure 18. An image of code showing the function which will store BLE devices containing "Spiritus Ludi" in the device name into an array.

```
72
73  static async scanForPeripherals() {
74    this.devices.length = 0;
75    return new Promise((resolve, reject) => {
76      setTimeout(() => {
77        this.sharedInstance.stopDeviceScan();
78        resolve(this.devices);
79      }, 10000);
80      this.sharedInstance.startDeviceScan(null, null, async (error, device) => {
81        if (device?.name !== null && device?.id !== null) {
82          if (device.name.includes('Spiritus Ludi')) {
83            console.log(device.name, device.id);
84            if (!this.devices.some((item) => item.deviceID === device.id)) {
85              this.devices.push({ deviceName: device.name, deviceID: device.id });
86            }
87          }
88        }
89        if (error) {
90          console.warn('Error in bleTest startdevicescan');
91          console.warn(error);
92          reject(error);
93        }
94      });
95    });
96  }
97
```

Provided these criteria are met, the last nested if statement on line 84 in Figure 18 shows the array which scanned devices are added to, is checked to ensure that the current scanned device doesn't already exist in the scanned devices array using the `some()` function. This prevents duplicates being rendered on the screen. If this returns `false`, then the unique device is added to the scanned devices array.

A `Promise` was used here instead of a `while()` function to avoid infinite loops and blocking functions, `Promise()` functions also allow the handling of asynchronous functions, and do not require a specific outcome or condition. It cannot be assumed that any devices will be successfully scanned, a set amount of time must pass regardless for the scan to be conducted. Using a `Promise()` will allow for a pending, fulfilled (`resolve()`), or rejected (`reject()`) outcome.

5.2.4 Connecting and Listening to Peripheral Devices

When the user selects the desired Spiritus Ludi sleeve to connect to, a function called `connectToDeviceAndListen()` in the `BleTest` class is called, seen in Figure 19. This function takes an argument known as `device` which receives the device name previously scanned.

The selected device is saved in the class property `deviceInfo`. Using `deviceInfo`, the corresponding `deviceID` is saved in the class property `connectedDevice` and then used to retrieve the BLE services and Characteristics using `discoverAllServicesAndCharacteristics()`. This will return all services and characteristics including those that have been programmed in by the peripheral manufacturers. Only the Spiritus Ludi Service and Characteristics are needed. To achieve this, `connectedDevice.services()` is called to assign the discovered services to `allSleeveServices` on line 107. `map()` is used on line 108 to iterate through each retrieved service. If the service contains the string `'3030'` then that will return the `service.uuid`. It is known the firmware holds three characteristics. Two which send and receive digital high-fives, and a third used to obtain the device serial number. The UUIDs are then assessed using `map()` to check if return `true` for `isReadable` or `isWritableWithResponse`. The characteristic UUIDs are then entered into their respective places in the object named `obj` on line 121. The object is returned containing the Characteristic UUIDs which are used for BLE communication once the connection is complete. This function connection phase is triggered in BTList.js on line 118.

Figure 19. This code shows the necessary part of the connection process to from the React Native application to the BLE Peripheral - The Spiritus Ludi Sleeve.

```
102  static async connectToDeviceAndListen(device) {
103    try {
104      this.deviceInfo = device;
105      this.connectedDevice = await this.sharedInstance.connectToDevice(device.deviceID);
106      await this.connectedDevice.discoverAllServicesAndCharacteristics();
107      const allSleeveServices = await this.connectedDevice.services();
108      const serviceUUID2 = allSleeveServices.map((service) => {
109        if (service.uuid.includes('3030')) {
110          return service.uuid;
111        }
112      });
113
114      this.serviceUUID = serviceUUID2[2];
115      const sleeveCharacteristics = await this.connectedDevice.characteristicsForService(
116        this.serviceUUID,
117      );
118
119      const sleeveCharacteristics2 = sleeveCharacteristics.map((chars) => {
120        const obj = {
121          toSleeve: '',
122          fromSleeve: '',
123        };
124        if (chars.isReadable === true) {
125          obj.fromSleeve = chars.uuid;
126        }
127        if (chars.isWritableWithResponse === true) {
128          obj.toSleeve = chars.uuid;
129        }
130        return obj;
131      });
132
133      this.tapFromSleeveChar = sleeveCharacteristics2[1].fromSleeve;
134      this.tapToSleeveChar = sleeveCharacteristics2[0].toSleeve;
135      this.connectedDeviceId = device.deviceID;
136
```

Once the connection has been successfully made, the function `listenForMessages()` is triggered shown in Figure 20. This function on line 172 in bleTest.js uses `monitorCharacteristicForService()` which accepts the arguments `serviceUUID`, and `tapFromSleeveChar` obtained in the connection process.

Figure 20. An image showing the code used to subscribe to the characteristic UUID which send data from the Spiritus Ludi sleeve to the Spiritus Ludi application, and how it handles that data. The `savetap()` function is used for user analytics only.

```
static async listenForMessages() {
  try {
    console.log('Now monitoring');
    this.connectedDevice.monitorCharacteristicForService(
      this.serviceUUID,
      this.tapFromSleeveChar,
      (error, characteristic) => {
        if (error) {
          console.log(`Characteristic monitoring error: ${error}`);
          return;
        }
        const value = characteristic?.value;
        const decodedValue = byteLength(value);
        console.log(`BLETest: Tap Received from connected sleeve: ${decodedValue}`);
        if (decodedValue === 1) {
          if (this.userName === null) {
            try {
              this.state = store.getState();
              this.userName = getUsername(this.state);
              console.log(`Username pulled from Store: ${this.userName}`);
            } catch (err) {
              console.log(`bleTest: Unable to get store: ${err}`);
            }
          }
          SignalRUtility.sendMessage(this.userName, JSON.stringify(decodedValue));
          saveTap({
            Username: this.userName,
            TimeTapSent: Date.now(),
            TimeTapReceived: null,
            TimeTapFailedSend: null,
            TimeTapFailedReceived: null,
          });
        }
      },
    );
  } catch (err) {
    saveTap({
      Username: this.userName,
      TimeTapSent: null,
```

This will now subscribe to that characteristic UUID and handle any incoming data from that UUID. In this case it is decoded from base64 to a byte array. If the byte array is equal to the integer 1, then the application will check the username is not null for usage analytics, then this will trigger the

SignalR function on line 199 to send the users digital high-five to the SignalR hub for distribution to the other team members. After the digital high-five has been successfully sent, the user interaction data is sent using the `saveTap()` function which triggers a HTTP POST request.

5.2.5 The SignalR Client Implementation

The SignalR service is made available through the `SignalRUtility` class in the singleton.js file. When this was created, there was only a requirement for a single SignalR team instance. Therefore, the hub connection string was hardcoded for this prototype to the variable `URL`.

The `constructor()` - Figure 21, sets the SignalR configuration using the `newHubConnectionBuilder()`. `.withURL()` takes the URL variable as its argument. `.withAutomaticReconnect()` will make four automatic reconnections if the connection is lost. This is the default bahviour. `.configureLogging()` provides logging at information level. The `.build()` method creates the connection based on configuration in the previous lines.

Figure 21. The connection builder with the URL redacted.

```
5
6   const URL = 'https:/[redacted]';
7   class SignalRUtility {
8     constructor() {
9       this.connection = new HubConnectionBuilder()
10        .withUrl(URL)
11        .withAutomaticReconnect()
12        .configureLogging(LogLevel.Information)
13        .build();
14
```

The method `connection.onclose()` is invoked when the connection is closed. The prototype implementation was created for the connection to be always open when the application was open. Therefore, in the event of the connection being closed, if it is not in a reconnecting state, then a new connection will be created using the `connection.start()` method. `connection.onreconnecting()` will be called when reconnection attempts start and `connection.onreconnected()` is called upon a successful reconnection attempt. This means it is less likely the user will need to intervene upon a dropped connection between the SignalR hub and the Spiritus Ludi SignalR client. `connection.on()` is invoked by the SignalR hub. As seen in Figure 22 on line 45, the hub method `'ReceivedMessage'` was invoked. This accepts the

arguments `user` and `message`. message is used as the argument to send the digital high-five from to the sleeve with `BLETest.tapToSleeve(message)`. connection.start() then starts the connection and resolves on success.

Figure 22. The methods inside the constructor that handle connection status and incoming messages from the SignalR hub.

```
15      //Flag to indicate if connection is reconnecting
16      this.isReconnecting = false;
17
18      // Subscription to the stateChanged event
19      this.connection.onclose((error) => {
20        console.log('Connection closed: ' + error);
21        // If the connection is not reconnecting, start a new connection
22        if (!this.isReconnecting) {
23          this.connection.start();
24        }
25      });
26
27      // Subscription to the reconnecting event
28      this.connection.onreconnecting((error) => {
29        console.log('Connection lost ' + error);
30
31        //Flag
32        this.isReconnecting = true;
33        //Notify
34      });
35
36      // Subscription to the reconnected event
37      this.connection.onreconnected((connectionId) => {
38        console.log('Connection re-established ' + connectionId);
39
40        // Set the flag to false
41        this.isReconnecting = false;
42        //Notify
43      });
44
45      this.connection.on('ReceiveMessage', (user, message) => {
46        console.log(`Tap received from ${user}`);
47        BleTest.tapToSleeve(message);
48      });
49
50      this.connection.start();
51    }
```

On line 64 - Figure 23, in FIGURE the function `sendMessage()` accepts user and message. This uses a try…catch statement to run `connection.invoke()`. This function will invoke the method

`'SendMessageOthers'` which exists on the SignalR Hub running in the cloud. This will catch any errors and present them as a warning in the console.

Figure 23. The client function that allows for digital high-fives to be sent to the hub which then will run the `connection.on()` method on other connected clients.

```
async sendMessage(user, message) {
  try {
    await this.connection.invoke('SendMessageOthers', user, message);
  } catch (err) {
    console.warn(`SingalR Singleton: Error sending tap to cloud: ${err}`);
  }
}
}
```

Errors are displayed in the console. There was no front-end design to notify the user if digital high-fives fail to send, so no further logic was implemented at this stage.

5.3 The Spiritus Ludi .NET Implementation

This simple implementation was created using Microsoft's documentation (Get Started with ASP.NET Core SignalR | Microsoft Learn, n.d.). This allowed for some initial testing of how SignalR performed over the web using web browsers. Using Visual Studio, the .NET 6 application was created and contains a class which allows for client server communication.

Figure 24. This class in TeamHub.cs allows for bi-directional communication between connected clients. It inherits from Hub, which is a base class for SignalR.

```
using Microsoft.AspNetCore.SignalR;


namespace TapRTC.Hubs
{
    1 reference
    public class TeamHub: Hub
    {//Inherits from SignalR hub, Manages connections, groups, and messaging
        //SendMessageOthers can be called by clients. Client.Others sends message to all
        //but the original sender
        0 references
        public async Task SendMessageOthers(string user, string message)
        {
            await Clients.Others.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

This method shown in Figure 24 allows for a client to send data to all connected clients but itself. This prevents the digital high-five being sent back to the origin. All digital high-fives received are all from other users and it will not wait for a response from the receiver. In the Program.cs file `using TapRTC.Hubs;` has been added to the top of the file and `var builder = WebApplication.CreateBuilder(args);` creates a new instance of the application builder. `Builder.Serfvices.AddSignalR()` registers the SignalR service via the dependency injection. This is required for SignalR to function. The web application is then built using `var app = builder.Build();` and `app.MapHub<TeamHub>("/teamHub");` maps incoming requests to the specified hubs.

This is then published to Azure. An Azure subscription and a resource for the application to be hosted in is required. Once these criteria are met, following the Publish process through Visual Studio will allow for the Application to be published to Azure and a URL will be created for the application.

6 Results

The results of the Spiritus Ludi system were deemed successful. The field tests were carried out with positive results and the software enabled the players to interact with each other during the field testing conducted by other project team members. The field tests found that there was an improvement of positive emotions with use of the Spiritus Ludi Sleeves among the teams using Spiritus Ludi (Jumisko-Pyykkö et al., 2023). User opinion from the field tests and associated analysis about the use of Spiritus Ludi can be found in the associated paper. Due to the success of the cumulative work of the Future Wearables team, Spiritus Ludi was featured at the 2023 Dutch Design Week (Hämeen Ammattikorkeakoulussa Kehitetyt Älyhihat Esittelyssä Dutch Design Weekilla - Hamkilainen Tutkimus, n.d.) – Finnish only, and gained attention from various media channels including the Consumer Trend Forecaster and Market Analysis Service WGSN (HAMK's Innovative Smart Sleeve Earns Growing International Recognition - HAMK, n.d.). The Spiritus Ludi project has gained interest from various parties, not only as an esports social mediated touch device, but also for the application of other use cases. At the time of writing this, Spiritus Ludi is currently listed for further development within the organisation pending further funding.

The use of SignalR allowed for persistent two-way communication over TCP connections, making the transfer of data consistently fast and reliable with its support for WebSocket protocol. The use of the SignalR Library simplified the real time communication implementation and can handle any transport method changes where and when necessary to enable an available real time communication service for as long as possible. The use of SignalR in this instance was one of the best available choices at the time due to its simplicity and its robust design to handle different communication methods without user input. Throughout the Spiritus Ludi project, SignalR has been consistently reliable and with low latency, however it is the opinion of the writer of this thesis that the SignalR Latency can be improved with further research and testing with other methods of SignalR deployment.

The BLE implementation proved to be challenging initially with the constraint of using a single processing thread on the peripheral side. C++ is generally considered a low-level language, therefore everything had to manually managed with the single processing thread. However, the firmware developed was able to adequately handle the user's needs, it was a consistently reliable speedy and implementation. Most issues that arose regarding the communication between the sleeve and the React Native Application were due to the fragility of the hardware. This was due the current limitations of today's available technology regarding flexible circuitry.

These readily available technologies in this project have provided a base platform for the implementation for social mediated touch. While focused on esports, it is not outside the realms of

possibility to apply this approach to other forms of social communication systems. The implementation of such software and hardware will have to be individually assessed. In some cases where one individual is controlling the actions of a device being physically worn by another individual or by a group of individuals, an Ethics Committee may have to be consulted to protect the rights and safety of those involved in its use, depending on the nature of the device being controlled – for example, a device designed for remote medical monitoring or administration.

The software was built on a prototype basis and was only originally designed to cater a six person esports team. However, as the project grew and more sleeves were made with improved physical designs, the software largely remained the same due to the time constraints of the project and the learning curve of the different software and systems required for the parts of the Spiritus Ludi system to work. While the software has been handling the demand of the demonstrations and field testing, there are some improvements and recommendations the author of this thesis would make in future development.

6.1 Bluetooth Implementation Improvements and Recommendations

The Bluetooth implementation could be further developed and tested. There is an intermittent bug when returning scanned devices. Occasionally, some of the peripheral devices returned are already paired to a central device. Therefore, they should not be shown as connectable. Currently it is not clear if this is a sleeve firmware or a React Native application bug. There are also capabilities within the BLE libraries which could be worth exploring for Spiritus Ludi, for example the Arduino BLE device can return the number of characteristics available, which can then be dynamically retrieved by the BLE central device.

The React Native application would benefit from more comprehensive error handling. This would require some additional UI/UX design for the user to be made aware of these errors in a way that does not unnecessarily affect users' concentration in the middle of a match.

Over the Air (OTA) Device Firmware Update (DFU) capabilities would be worth investing resources in developing into future iterations. Currently the sleeve firmware must be manually updated individually. Future iterations should include OTA DFU capabilities, especially if Spiritus Ludi heads towards a production state. This will require additional development of the firmware, and the Spiritus Ludi Application will need to have the capability check the current firmware version and offer the firmware update to the users when appropriate.

Utilising more of the applicable BLE libraries on the peripheral and central sides would enrich the BLE service and create a robust short range communication service between the sleeve and

mobile handset. Lastly, future BLE implementation with Spiritus Ludi should consider further security measures. The pairing method used in this protype was suitable in this case, however it can be vulnerable to a Man in the Middle (MITM) attack. Passkey entry would offer a suitable level of security at this stage, but the current design limits this being integrated due to the lack of a user inface that can display data such as a four- or six-digit pass code on the sleeve, either the code could be programmed and stored into sleeve on first start up, which will require a recovery system to be created. Another option would be to consider Out of Band Pairing system, using communication such as NFC to exchange pairing data. However, this may be unnecessary for such a system not transferring personal or sensitive information.

6.2 SignalR Implementation Improvements and Recommendations

The SignalR service has been extremely reliable, only limited by factors such as the connection strength to the Wi-Fi or cellular network, or the Nokia G10 Handset itself as it has been slow to respond when navigating the operating system with no other applications open.

As mentioned previously, there is currently only one instance of SignalR running. For future iterations, dynamically created SignalR Hub Instances should be created to enable multiple team communication. These team instances will need to be managed appropriately to ensure server-side resources are not unnecessarily wasted. This feature implementation will require further UX/UI and software planning and designing, including addressing subjects such as how the team sessions will be uniquely identified, who in the esports team will be able to create the team instance for the whole team, what kind of administration privileges would be needed to manage the connected users and who would get those privileges, and what secure method will be implemented to allow authorised users only into the team. The team instancing would provide great value to the Spiritus Ludi System.

Error handling could be improved here as well. Currently the user can see what connection state the cloud is in, however if there is an issue with the client side or the server side, there is no way to tell through the application alone. There is currently a fault reporting system that allows a user to report issues with the application or sleeve through the application which leads to a Microsoft forms form, suitable for the current implementation, however going forward, an organic fault reporting system would be beneficial where reports could be made with accompanying log files or error reports from the sleeve. Any communication alterations between the sleeve and the Spiritus Ludi application should not negatively impact the BLE latency.

As with the BLE implementation, the SignalR implementation would also benefit from its other capabilities. It is also worth noting that should the demand of the Spiritus Ludi system increase dramatically, then the backend approach should be revisited to decide the best way to scale out.

6.3 Testing

Some structured formal testing practices would have been beneficial during the development of this iteration, however due to time constraints, testing was done on an ad hoc basis. To appropriately validate the software's functionality and performance, appropriate tests should be created for all three sparts of the Spiritus Ludi system.

For testing of the end-to-end latency from user interacting with the touch pad to the last person receiving the digital high-five, it is likely an external system would have to be created to create the datapoints. Currently datapoints are being created in the Spiritus Lud application where the application receives a digital high-five from the sleeve and after the application has sent the digital high-five to the sleeve of the other connected SignalR clients. For accurate latency testing, and end to end test should be developed, and smaller tests within the firmware, React Native Application, and the .NET SignalR service to determine if there are any choke points or inefficient processes slowing the transfer of data down.

7 Summary

The research questions were answered by firstly looking at current core protocols and their characteristics. This showed that WebSocket technology which is layered onto the TCP protocol allowed for a data safe persistent connection. Microsoft developed SignalR which removed the complexities of utilizing WebSocket technology and provided a graceful fallback to other transport methods should the WebSocket option not be available. This settled that SignalR was well suited for our needs. SignalR was then applied into the software implementation, used to handle communication between connected clients which were running in purpose developed phone applications. With the BLE implementation handling communication between the sleeve and the handset, and SignalR handling communication over the internet between all connected clients, this provided a low latency social mediated touch software system from end-to-end from sender to all recipients. With the characteristics fulfilled through the correct selection of software, the field tests and associated research was able to be successfully carried out.

This project has taught me many things, including the importance of planning in a large project, given me insight into the complexities of developing software to be run in constrained environments where capabilities previously taken for granted such as dynamic arrays are not possible. The implementation of WebSocket technology has given me valuable experience in understanding and working with different transport protocols and why it is important to select the current protocol for a given task. Working in this multidisciplinary team has also shown me the importance of understanding my own field so I can effectively communicate what is and isn’t possible, and capabilities are possible, what realistic time frame should be expected to implement such things.

The work with Spiritus Ludi and the Future Wearables project is expected to continue. Wearable technology is a field that has seen a lot of development in recent years, and the interest that Spiritus Ludi has generated from various demographics has shown development is likely to continue.

References

Belshe, M., Peon, R., & Thomson, M. (2015). *Hypertext Transfer Protocol Version 2 (HTTP/2)* (Request for Comments No. RFC 7540). Internet Engineering Task Force. https://doi.org/10.17487/RFC7540

Bluetooth low energy Services, a beginner's tutorial—Bluetooth low energy—nRF5 SDK guides—Nordic DevZone. (2015, August 26). https://devzone.nordicsemi.com/guides/short-range-guides/b/bluetooth-low-energy/posts/ble-services-a-beginners-tutorial

Bluetooth permissions | Connectivity | Android Developers. (n.d.). Retrieved 6 October 2024, from https://developer.android.com/develop/connectivity/bluetooth/bt-permissions

Bousquet, J., & Ertz, M. (2021). *eSports: Historical Review, Current State, and Future Challenges* (pp. 1–24). https://doi.org/10.4018/978-1-7998-7300-6.ch001

Get started with ASP.NET Core SignalR | Microsoft Learn. (n.d.). Retrieved 19 October 2024, from https://learn.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-6.0&tabs=visual-studio

Getting Started with XIAO nRF52840 | Seeed Studio Wiki. (n.d.). Retrieved 12 October 2024, from https://wiki.seeedstudio.com/XIAO_BLE/

Hämeen ammattikorkeakoulussa kehitetyt älyhihat esittelyssä Dutch Design Weekilla—Hamkilainen tutkimus. (n.d.). Retrieved 19 October 2024, from https://blog.hamk.fi/hamkilainen-tutkimus/hameen-ammattikorkeakoulussa-kehitetyt-alyhihat-esittelyssa-dutch-design-weekilla/

HAMK's innovative smart sleeve earns growing international recognition—HAMK. (n.d.). Retrieved 19 October 2024, from https://www.hamk.fi/en/hamks-innovative-smart-sleeve-earns-growing-international-recognition/

Introduction · React Native. (n.d.). Retrieved 2 October 2024, from https://reactnative.dev/docs/0.71/getting-started

Jumisko-Pyykkö, S., Halme, A., Hattingh, G., Kerkola, H., Scotcher, B., Rapp, A., Rönkä, W., Saarinen, J., Salo, T., & Salonen, T. (2023). *Spiritus Ludi: Smart Wearable Textiles that Foster and Elevate Connections in E-sports Teams* (Internal)

Melnikov, A., & Fette, I. (2011). *The WebSocket Protocol* (Request for Comments No. RFC 6455). Internet Engineering Task Force. https://doi.org/10.17487/RFC6455

@microsoft/signalr—Npm. (n.d.). Retrieved 2 October 2024, from https://www.npmjs.com/package/@microsoft/signalr/v/7.0.2

Oksala, A. (2022, June 14). *The importance of communication in esports: An ethnographic interview with a Finnish Counter-Strike: Global Offensive team* [Pro gradu -työ]. Laturi.Oulu.Fi. https://oulurepo.oulu.fi/handle/10024/21016

Overview of ASP.NET Core SignalR | Microsoft Learn. (n.d.). Retrieved 20 October 2024, from https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-8.0

react-native-ble-plx—Npm. (n.d.). Retrieved 2 October 2024, from https://www.npmjs.com/package/react-native-ble-plx/v/2.0.3

react-native-device-info—Npm. (n.d.). Retrieved 2 October 2024, from https://www.npmjs.com/package/react-native-device-info/v/10.3.0

react-native-permissions—Npm. (n.d.). Retrieved 2 October 2024, from https://www.npmjs.com/package/react-native-permissions/v/3.6.1

react-native-quick-base64—Npm. (n.d.). Retrieved 2 October 2024, from https://www.npmjs.com/package/react-native-quick-base64/v/2.0.5

react-redux—Npm. (n.d.). Retrieved 2 October 2024, from https://www.npmjs.com/package/react-redux/v/8.0.5

Remote Procedure Call—IBM Documentation. (n.d.). Retrieved 20 October 2024, from https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call

RFC 9293: Transmission Control Protocol (TCP). (n.d.). Retrieved 15 September 2024, from https://www.ietf.org/rfc/rfc9293.html#name-key-tcp-concepts

Spiritus Ludi—Spirit of the Game in Esports—HAMK - Häme University of Applied Sciences. (n.d.). Retrieved 9 November 2023, from https://ddw.nl/en/programme/10006/spiritus-ludi-spirit-of-the-game-in-esports

Use hubs in ASP.NET Core SignalR | Microsoft Learn. (n.d.). Retrieved 20 October 2024, from https://learn.microsoft.com/en-us/aspnet/core/signalr/hubs?view=aspnetcore-6.0

WebRTC API - Web APIs | MDN. (2023, November 9). https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API

Wood, R. T. A., Griffiths, M. D., Chappell, D., & Davies, M. N. O. (2004). The Structural Characteristics of Video Games: A Psycho-Structural Analysis. *CyberPsychology & Behavior*, *7*(1), 1–10. https://doi.org/10.1089/109493104322820057

Woolley, M., & Anees, I. (2024, March 15). *The Bluetooth® Low Energy Primer*. https://www.bluetooth.com/. https://www.bluetooth.com/bluetooth-resources/the-bluetooth-low-energy-primer/?utm_source=internal&utm_medium=blog&utm_campaign=technical&utm_content=the-bluetooth-low-energy-primer

Appendix 1: Material Management Plan

During the development project, the software created for Spiritus Ludi was stored in the following four internal company GitHub repositories:

https://github.com/hamk-uas/SpiritusLudi_RN
https://github.com/hamk-uas/TapApp_Sleeve
https://github.com/hamk-uas/TapStorage
https://github.com/hamk-uas/TapApp_.Net

It is likely there will be further development in the mentioned repositories. The current states of the repositories are kept in the version history.

These repositories are kept in line with HAMK's Data Protection Policy.