Skitracker: Measuring Skiing Performance using a Body Area Network

Thomas Homewood
Razee Hussein-Jamal

Institutionen för informationsteknologi
Department of Information Technology
Abstract

**Skitracker: Measuring Skiing Performance using a Body Area Network**

*Thomas Homewood and Razee Hussein-Jamal*

Cross-country skiing is a technically demanding sport that requires the use of several distinct techniques. Typical research in this area involves testing in laboratory conditions using bulky equipment.

The goal of this project is to create a complete system that allows tests to be conducted on skiers in real conditions, using a network of wireless sensors to collect data.

We begin by outlining what data coaches and skiers want to see, before describing in detail our system, which comprises a set of networked wireless sensors mounted on the skier, a smartphone used as a base station, and a web interface viewing and editing recorded data.

We then conduct testing of the system on Swedish national team skiers, and use the data collected to compare the relative performance of these skiers in outdoor conditions. Finally the system is evaluated and future developments are discussed.
Contents

1 Introduction 7

2 Technique and technical aspects of cross-country skiing 9
   2.1 The importance of technique ................................. 10
      2.1.1 Classical ............................................. 10
      2.1.2 Skating .............................................. 10
      2.1.3 Investigating technique ................................ 12
   2.2 Physiological performance .................................. 13

3 Implementation 14
   3.1 Equipment Used ........................................... 15
      3.1.1 Sensors .............................................. 15
      3.1.2 Smartphone gateway .................................. 16
      3.1.3 Web server: .......................................... 16
   3.2 Android .................................................... 17
      3.2.1 Introduction .......................................... 17
      3.2.2 A brief overview ...................................... 18
      3.2.3 Application lifecycle and persistence .................. 19
      3.2.4 Collecting data from Bluetooth sensors ................. 21
      3.2.5 Handling sensor data .................................. 22
      3.2.6 Multithreading and periodic execution ................. 25
      3.2.7 User Interface ......................................... 28
   3.3 The Track API ............................................... 37
      3.3.1 Introduction .......................................... 37
      3.3.2 Architecture .......................................... 37
      3.3.3 Lifecycle ............................................. 38
   3.4 Web interface ............................................... 39
      3.4.1 Introduction .......................................... 39
      3.4.2 Features .............................................. 39
   3.5 Shimmer sensor nodes ..................................... 46
      3.5.1 Introduction .......................................... 46
      3.5.2 TinyOS and nesC programming language ................. 46
      3.5.3 Design and Implementation ............................ 47
   3.6 System overview .......................................... 49

4 Testing 50
   4.1 Testing the system ......................................... 51
   4.2 Data analysis ............................................. 53
      4.2.1 Women’s skate race .................................... 56
4.2.2 Men’s skate race ................................. 61

5 Evaluation ................................. 65
  5.1 Performance of the system ................................. 66
  5.2 Future work ........................................ 67
  5.3 Division of work .................................... 68

6 Appendix ......................................... 71
  6.1 The Track API: A user’s guide ............................. 72
    6.1.1 Introduction .................................. 72
    6.1.2 Setting up the track API .......................... 73
    6.1.3 Using the accounts layer .......................... 74
    6.1.4 Using the data layer .............................. 74
    6.1.5 Using the visual layer ............................ 94
    6.1.6 Displaying maps ................................. 95
    6.1.7 Displaying line graphs ............................ 97
    6.1.8 Displaying scatter graphs ........................ 99
  6.2 Shimmer: An architecture description ..................... 101
    6.2.1 Introduction .................................. 101
    6.2.2 Interfaces used in the application ................. 102
  6.3 Android: Bluetooth communication ......................... 104
    6.3.1 Introduction .................................. 104
    6.3.2 Connecting to Bluetooth sensors ................. 104
    6.3.3 Communicating with Bluetooth sensors ............ 107
Chapter 1

Introduction

Cross-country skiing is a highly demanding competitive sport with a strong emphasis on both technique and technology. It is one of the few sports that requires a competitor to use several distinct techniques in one event, chosen based on factors such as gradient, speed and snow conditions.

Traditionally, the technological aspect of the sport has revolved around the development of new materials and construction techniques for the skis and other equipment used. However, more recent advancements have come about through using technology to monitor physiological and technical parameters and using the data collected to suggest ways to improve the end result.

Typically this monitoring is done indoors in a lab environment, using roller skis on a large treadmill that is about 3 metres square. A skier’s technique is analysed using high-speed cameras positioned at various angles around the treadmill, and physiological parameters such as heart rate and oxygen uptake are recorded using sensors attached to static lab equipment.

However, this indoor testing has its limitations. Firstly, roller skis on a rubber treadmill behave differently to standard skis on snow, which affects the technique of the skier by some degree. Secondly, environmental conditions are vastly different outdoors, and can change rapidly, with temperatures typically between -30 and +10 degrees Celsius, compared to a lab at room temperature. Other factors such as humidity and wind speed also become relevant when skiing outdoors. For these reasons, it is desirable for both coaches and skiers if performance data can be collected whilst skiing outdoors on a typical track, using all the standard equipment. Furthermore, the equipment used in the lab tends to be connected by cables to data loggers, meaning it takes a long time to set up the equipment ready for use.

With this in mind, it was decided that a new approach to collecting data was needed, one that uses equipment that can be used both indoors and out, and does not require a lot of effort to set up. The bulky, wired lab equipment would be replaced by a Body-Area Network (BAN) of small wireless sensors attached to the skier. Instead of high-speed cameras, accelerometers in either a smartphone or a sensor node would be used to capture the movement of the skier on three axes, and wired physiological sensors would be replaced by a wireless sensor belt worn by the skier on their chest.

This project has been run in conjunction with the Swedish Institute of Computer Science (SICS), who were responsible for developing the software that analyses the skier’s technique using accelerometer data. Our task was to develop a complete system, from the network of sensors used for data collection, to the presentation of the data to the user. With respect to the technique analysis part, our system would collect the accelerometer
data from a sensor and pass it to the analysis software, which would then produce output that forms part of our presentation interface.

This report will first look at the technical aspects of cross-country skiing, with a focus on what data is important to the coaches and skiers, before moving on to describing the system we have created in detail, and finally looking at how the system performed when tested on elite skiers. Also included is a technical report which uses data collected from this testing to study how different skiers perform during a race. Finally the system will be evaluated and future advancements discussed.
Chapter 2

Technique and technical aspects of cross-country skiing
2.1 The importance of technique

Cross-country is quite unusual among endurance sports in that there exist several different techniques that skiers may use. Broadly speaking, there are two styles that are commonly used by both amateurs and professionals: the classical style and the skate style.

2.1.1 Classical

Regarded as the traditional style and used for many hundreds of years, this involves skiing on a track consisting of two parallel grooves cut into the snow [6]. Propulsion comes from the fact that the skis are curved, with the central section sitting higher than the ends. This underside of this central section is designed to grip the snow, by means of either a pattern engraved into it, or a coating of adhesive wax (known as grip wax). When a skier distributes their weight evenly across both skis, the curvature of the skis is sufficient to lift the more adhesive central portion of the skis clear of the snow surface, allowing the skier to glide in the grooves. When the skier places their weight on only one ski, it is depressed and makes contact with the snow, allowing the skier to kick backwards, and are thus propelled forwards.

Within this style, several techniques may be used, depending on the skier’s speed and the type of terrain:

- Herringbone
   When travelling uphill, the herringbone technique is used as a way to exert more power, at the expense of speed. This involves forming a ‘V’ pattern with the skis and pushing backwards with each kick. This method prevents the skier from losing ground due to sliding backwards [6].

- Diagonal stride
   This technique is used when skiing on the flat, and is analogous to running, with the kick and glide performed on alternate sides. The poles are planted on the opposite side to the ski being pushed backwards, which again is the same as the arm movements used when running [5].

- Double poling
   This technique uses only the arms for propulsion, whereby the skis are not kicked but held together in parallel, with both poles moving in unison. Extra pole force may be achieved by performing a small jump immediately before planting the poles in the snow, in order to increase the impulse upon impact and thereby increase the acceleration achieved [6].

Several variants of the above techniques exist, however they can largely be classified in these groups. It should be noted that there also exist techniques for turning, which typically involve leaning or gliding on the ski that is on the inside of the turn, and kicking repeatedly on the outside ski. The ski poles may also be used to help effect a turn.

2.1.2 Skating

As its name suggests, this style of cross-country skiing resembles ice or roller skating, with the forwards movement derived from pushing the skis sideways away from the body. Consequently, this does not require pre-existing grooves in the snow.
Although also very old, it only became popular in the early eighties, when skiers competing in the ‘Vasaloppet’ (an 87km annual long-distance race held in Sweden) began to use a skating motion towards the end of the race. They did this because the length of the race meant that towards the end the grip wax on their skis would wear down and become less effective. They countered this by using a skating motion for forwards propulsion, which was found to be more efficient under these conditions. Several skiers, among them the famous American Bill Koch, took this technique (known at the time as marathon skating), and developed it over the next five years into what is now known as skate skiing [7].

There are five techniques within the skate style, with each technique suitable for use on a certain terrain. In 1996, Holmberg [11] classified these techniques, referring to them as gears, as a skier will change between techniques depending whether they are travelling uphill, downhill, or on level ground, much like the driver of a car uses a gearbox to select the most suitable gear.

1. Gear 1 - Diagonal V
   Primarily used for very steep uphill gradients, this gear is similar to the Herringbone technique found in classical cross-country skiing, but with a short glide after each push on the skis [4]. This is rarely seen in competitions, as most elite skiers are able to climb even steep gradients using gear 2.

2. Gear 2 - V1
   This technique is used for skating uphill, and utilises an asymmetric double poling motion (the two poles are planted sequentially), with initial pole plants coinciding with every other leg push [14]. As the poling occurs on the same side repeatedly, this gear has two forms, left and right, which may be used interchangeably by the skier depending on various factors. Typically, when cornering in gear 2, skiers will use the right form when turning left and vice versa, as this allows them to use the asymmetric nature of the poling and leg movements to effect the turn.

3. Gear 3 - V2
   Used on moderate inclines and level terrain [14], this gear is similar to gear 2 but with pole plants on every leg push, thus allowing greater speed to be achieved. Strong skiers may be able to use gear 3 on steeper inclines.

4. Gear 4 - V2 alternate
   Again a development of gear 2, with the difference being that the poling motion is symmetrical, which provides a more concentrated period of acceleration compared to gear 2. This is used on level terrain [14].

5. Gear 5 - V skating
   This gear is simply a skating motion using only the legs, and is used only on downhill sections where propulsion from the ski poles is not required [14].

According to some definitions, there exist a couple of other gears that are used for curves and for travelling downhill without even moving the legs, however, these are not as frequently used as the techniques detailed above.
2.1.3 Investigating technique

Classical style

Within the classical style, the main area of interest is the poling technique, as this provides the majority of the propulsive force when skiing using double poling, and is also important when using the diagonal stride technique. Current research in this area is looking in detail at both the force exerted through the poles and the poling frequency. To measure the pole force, a load cell is placed inline with the carbon-fibre pole immediately underneath the handle, mounted in an aluminium tube [15]. This load cell is thereby subject to the resultant ground reaction force that occurs when the pole is planted on the ground and force exerted upon it. Data derived from these measurements includes the time to peak force, impulse force, and average cycle force.

Although relatively lightweight for a sensor (the load cell and tube weigh 135g combined [15]), this equipment does add considerable weight to the ski pole, which is a disadvantage in a sport where much time and effort is spent reducing the weight of the equipment used. Furthermore, this sensor is connected via a cable to a data logger placed on the skier’s back, which itself is quite bulky. The cable is run down the arm and attached in several places, meaning that the system takes a long time to set up.

Therefore, after consultation with the Swedish Winter Sports Research Center, where this existing equipment is used for experiments, it was decided to investigate the possibility of developing an alternative system, where the data logger is removed, and the sensor readings are transmitted directly to a base station, such as a mobile phone. This would also remove the need for a cable to be run up the skier’s arm. Additionally, our investigation would look at the possibility of replacing the load cell, and its aluminium housing, with something more lightweight.

Skate style

Analysis of skate skiing also involves the study of poling technique, but additionally focuses on gear selection, which is the study of what gears a skier uses at certain times during a race. Andersson et al [8] observe that during a skate time trial of 1.43km, 29.1 gear changes were made, which equates to 203 changes in technique over the course of a 10km race of the same nature. Although a longer distance race might not have quite so many changes, many coaches take the view that the eventual winner of such a race may be the skier who changed gears at the most suitable times.

With this in mind, SICS have a working prototype of a system that is able to classify which gear a skier is using, as well as provide other performance-related data such as the consistency and symmetry of their motions, using data collected from an smartphone’s accelerometer. The device is mounted on the subject’s chest, close to their centre of mass, and is sampled with a frequency of between 50 and 100Hz. This data is then run through their analysis tool after the recording session, which is able to generate reports that may be imported into data analysis software.

One of the goals of this project will be to integrate this analysis tool into a larger system, so that the technique data it generates can be viewed in context alongside other performance indicators such as heart rate, speed and others. This would provide coaches and skiers alike with a combined set of data that allows them to see, for example, which gears are used at certain points on a track. This would help them to see which gear changes were beneficial to the skier, and which ones were detrimental.
2.2 Physiological performance

Physiological parameters such as heart rate and breathing rate are closely monitored by skiers and coaches alike. During training sessions, skiers will almost always wear a heart rate monitoring belt on their chest, as their performance relative to their maximum optimal heart rate is critical. Simply put, if this maximum is reached or exceeded, then their body will begin respiring anaerobically, producing lactic acid as a result, making it very hard or impossible for the skier to continue.

When conducting experiments indoors on the treadmill, the breathing rate, oxygen uptake rate and other respiration-related parameters may be recorded using a mask, placed over the skier’s mouth and nose, connected via a tube to oxygen and carbon dioxide analysers. This method gives a very accurate picture of how the subject is performing, but is not suitable for use outside due to the equipment’s large size.

This project will aim to incorporate sensors that can measure both heart and breathing rate, thus allowing the relationship between variables such as speed and elevation, technique characteristics such as consistency and symmetry, and physiological performance, to be investigated. Our system will use either off-the-shelf hardware or a custom-made sensor to record this data, depending on what already exists on the market.
Chapter 3

Implementation
3.1 Equipment Used

The Skitracker system comprises several components:

3.1.1 Sensors

A set of sensors used for capturing various parameters from both the skier and their equipment. These sensor devices communicate with a mobile device using Bluetooth. Each sensor has a Bluetooth name that contains a two-character Sensor Identification Code (SIC) that is used to determine which type of sensor it is. These include:

- Zephyr™ BioHarness:
  A sensor belt capable of recording a wide range of physiological data such as heart rate, (as well as variability and ECG), breathing rate and amplitude, estimated core temperature, and galvanic skin response (a measure of how moist the skin is). A three-axis accelerometer is also included. The device actually comprises two sections: a Bluetooth sensor node containing the electronics, and a washable belt that is worn around the subject’s chest, which contains conductive sensor pads used for picking up heart rate signals. The sensor node is simply pushed onto the belt and connection is made via three docking pads on the belt.
  Alternatively, the sensor node may also be used with a smart shirt. This has the same attachment mechanism as the belt, and supports recording of the subject’s heart rate. The main advantage of this shirt over the belt is that of comfort and convenience, as it is very similar in design to the skin-tight vests normally worn by the skiers.

- Pole force sensor:
  This custom-made sensor is used for wirelessly recording the force exerted on a ski pole. After discussion with several skiers, it was decided to measure the force exerted through the wrist strap attached to the pole. The reason for this is that elite skiers do not grip the poles tightly when poling, instead they place most of the force in their wrists and pull on the strap, whilst lightly holding the pole to keep it in the correct place. The idea therefore was to capture the tension in the wrist strap using a load cell inserted inline with the strap, as shown in figure 3.1.
  The load cell chosen (LSB200 by Futek, California) has an operational range of ±220N (in both tension and compression), and is around 17mm square, making it ideal for our application. The signal from the load cell is connected to a strain gauge amplifier (XN3 by TeXense, Indiana) in order to amplify the signal we receive to the standard 0-5V level. The signal is then fed to a sensor node (Shimmer 2 by Shimmer Research, Dublin) which samples at a rate of 50Hz and sends readings to the base station for processing.
3.1.2 Smartphone gateway

In our system, an Android mobile device works as a gateway between wireless sensors and a remote server. The Android application we have implemented is compatible with Android version 2.1 and higher, and handles communication with both the wireless sensors and the back-end server. Additionally, the application collects data from the smartphone’s internal accelerometer and GPS receiver. Communication with the server is handled using our bespoke Track API (see section SECTION for details).

As well as collecting and transmitting data, the application displays live data to the user in an easy to use interface, and also allows a user to monitor another skier in real-time.

3.1.3 Web server:

Both the web UI and Track API run on a remote server which is used for all file and database storage. This server is a standard PC with a fast CPU, which is beneficial when running the technique analysis software, and runs the standard Linux, Apache, MySQL and PHP (LAMP) stack. Using the Track API.
3.2 Android

3.2.1 Introduction

Our system uses a smartphone as both a gateway between our sensors and the cloud, and as an instant feedback medium, giving the skiers information about their progress.

Android was chosen as the platform on which to develop our application as it widely available across a large range of devices, with a large market share [1], and it supports Bluetooth version 2.1, which is the de facto standard for many commercially-available mobile sensors. The other main mobile operating system, iOS, currently does not support serial data transmission over Bluetooth, meaning that at the moment it is not suitable for our purposes. As new sensors begin to use Bluetooth 4 more and more, we expect this to change, meaning the development of an iOS application will become possible.

Android uses standard Java as a development language, albeit with some libraries specific to Android added, so our previous experience with this language enabled us to get up and running quickly.

Our application is suitable for use on any device running Android version 2.2 or higher. It does not require any particular hardware in order to run, however Bluetooth support, a GPS receiver and an on-board accelerometer are preferred. In this section we will give a brief overview of the application, before moving on to looking at the issues encountered during development and how they were solved, before finally focusing on the user interface design and user experience.
3.2.2 A brief overview

To use the system, the user must login to the application with either a Skitracker or Facebook account. In the case of the former, the user can create an account on the device, which can also be used in the web UI. The option to use a Facebook account to login is provided as it is a fully automated process for the user, requiring only that they are logged into Facebook on their device before attempting to connect, and that they grant Skitracker access to data from their Facebook account. If access is permitted, then the application collects user data automatically and uses this to create an account on the server, in much the same fashion as a normal Skitracker account.

The application supports the use of multiple servers, so the user may choose in the settings which server they wish to connect to. The list of available servers is downloaded from a “master server”, the address of which is hard coded and does not change. During development, this allowed us to develop and publish new code on one server, whilst keeping a more stable server for skiers to use for testing.

To ensure that the user always has the latest version of the application, we have implemented an automatic update system. When launched, the application checks in a version repository on the server and downloads and installs an update if one is available. For a full-scale production application that is available on Google’s Play Store, this updating is handled there automatically, but since our application is not available to the general public, a bespoke update mechanism is required.

Upon launching the application, the user is presented with a tabbed user interface, where the content of each tab is used to show different information. The tab initially displayed shows the Activity feed, which displays details of any live tracks currently in progress, and past tracks that have already been completed. Clicking on a live track will take the user to a screen where they can monitor the track in real-time, whilst clicking on a past track will launch an interface where the track’s data may be viewed in graph form.

The second tab, entitled My tracks, allows the user to view and edit tracks they have recorded on the device. The tracks are presented in a simple list, sorted in reverse chronological order. Upon clicking on a track, the user may perform basic administrative operations, such as renaming, deleting, or uploading the track to the server. If the track exists on the server, they may also view its data graphically, in much the same way as is possible in the web UI. The presentation of this data is accomplished using the commands available in the track API, which is described in section 3.3.

The third tab contains the user interface for starting and stopping tracks, as well as for viewing live data. The screen is divided into two sections, the first displaying a details of one variable, for example heart rate, in a large font that is easy to read. The user may change the variable displayed in this section simply by touching it, whereupon they are presented with a list of variables to choose from. The user can also change this section to display a progress bar that represents the value of the variable with respect to a minimum and maximum value. The minimum and maximum values can be set at any time, and if the value of the variable changes so that it lies outside these bounds, the text in the section turns red to alert the user. The second section can contain either a map showing the skier’s progress, or a set of six smaller boxes that can display any six variables.

In portrait mode, these two sections appear one below the other, whilst in landscape mode they are displayed side by side. This is achieved having a layout definition file for each orientation, which allows the Android OS to choose the appropriate layout automatically. All elements in this tab are refreshed once every second whilst a track is running.

The fourth and final tab is used for remote monitoring of an active skier. It functions in exactly the same way as the previous tab, in that the user may choose which data to
view, and optionally display a map. Data is retrieved periodically from the server using the Track API.

The user interface is discussed in greater detail in section 3.2.7, where screenshots are also provided.

### 3.2.3 Application lifecycle and persistence

Being an OS designed to run on compact devices with limited hardware resources, Android imposes some constraints on how applications may run, in the form of an Activity Lifecycle, where an Activity is a ‘single, focused thing that the user can do’ [10]. This lifecycle stipulates that once created and started, an activity may interact with the user, but more importantly, it states that an activity may be destroyed at any time by the OS if it is using too much CPU time or system memory, or indeed if the user simply locks the device. Additionally, when the device enters sleep mode (characterised by a lower CPU frequency and optionally a disabled screen and/or wireless radios), activities are often paused, stopped, or destroyed. A full description of the activity lifecycle is available at [10].

For this reason, using an Activity is not the ideal way to collect data for long periods of time, during which the device will go into sleep mode and almost certainly be locked by the user. For applications that need to run in the background for long periods of time, Android provides another system building block: a Service.

Services in Android are intended for use by applications that need to run simple background tasks without direct user interaction, and are well suited to our application’s needs. They allow us to collect sensor data at high sampling frequencies (100Hz) during a ski run lasting several hours. However they come with one caveat: A service may not directly interact with the user in any way. This Activity-Service paradigm makes a great deal of sense in any environment, but is particularly important in a mobile OS where resources are at a premium but applications and widgets require regular updating. That said, it does make application design trickier, due to the fact that our application must handle user interactions in an Activity, but perform the majority of the data collection and processing in a Service. It should be noted that a Service does not run in a background thread, as it is not designed for running code that blocks for long periods of time, rather for tasks that perform operations that do not require user interaction.

One approach for communicating between an Activity and a Service is to use static methods in that can be called from any other class. However, this requires the creation of a large number of static variables, and static objects can lead to higher memory usage, which is not desirable. The method we use involves using a Handler to act as the go-between between the Activity and the Service. The Activity, containing a Handler object, is created when the application is launched, and subsequently creates the Service, passing to it a reference to the Handler. The Service is then able to send messages with various unique IDs back to the Activity. Typically, each message I.D. represents an event, for example a track was started, or connection to a sensor was lost. Messages can also contain bundles of variables if required.

One final thing to note about a Service is that even though it is intended for long-running background execution, the OS may destroy it if its resources are required. To circumvent this, our Service acquires a WakeLock from the OS when it is started, which prevents the device entering a low power state, but allows the screen to turn off. This ensures that the device will not attempt to destroy the Service. Whilst this solution will negatively affect the battery life of the device whilst running our application, the effect is not that great, due to the fact that our application is already using GPS, Bluetooth
and optionally GPRS / HSDPA for data transmission, all of which are known to be power hungry. The WakeLock is released when the Service is stopped.

Because we use a Service for collecting data from sensors, the user may exit the app, using the home or back button, at any time during the track without affecting the recording or transmission of data. When the Activity is re-launched, it re-establishes the connection with the Service automatically.
3.2.4 Collecting data from Bluetooth sensors

The Skitracker system comprises several Bluetooth devices communicating to each other via Bluetooth version 2.1. The android platform device which functions as a gateway, supports Bluetooth technology and the application framework access to Bluetooth functionality through the Android Bluetooth APIs [9]. The Android platform mobile device acts as a master, requesting for connection to the slave devices. For security reasons, the application must request permission from the OS to use the Bluetooth subsystem.

To establish a serial connection, the application is configured to use Bluetooth with a special Unique User ID (UUID) that indicates to the OS that it wishes to use the Serial Port Profile (SPP) method of data transmission. Once the system has checked if the Bluetooth is turned on, it starts an asynchronous scan for devices, which takes approximately 12 seconds to complete. During this scan, the application receives events from the OS when a new device is detected. The name of this device is examined, to see if the first two characters match any of the pre-defined SICs. If a match is found, the user is given the option to pair with and connect to the sensor, and this association is preserved even when the application is stopped, meaning that the user does not have to re-pair or re-connect the next time the application is used.

Recording data from a sensor is handled by a background thread, as the process of actually reading the data from the socket will block execution until the entire message has been read. For this reason, each connected sensor is given its own thread. When a message is received, it is sent to a Handler object located in the background service, accompanied by a message code that identifies where the message came from, and the current timestamp.

In short the complete lifecycle for communicating with a Bluetooth sensor is as follows:

1. Discover local Bluetooth devices
2. Pair to a discovered Bluetooth device
3. Connecting to this Bluetooth device via a Bluetooth socket
4. Accepting incoming and outgoing connection
5. Receive data periodically
3.2.5 Handling sensor data

As mentioned above, Skitracker acts as both a gateway for sensor information between the sensors and the cloud, and as a tool which can be used for live monitoring of a skier’s run (either on the skier’s own device or on a device belonging to a third party). Additionally, Skitracker also collects sensor information from the device itself and transmits this to the server, meaning that the device is not simply a gateway, but also a sensor node.

As the skier may often be out skiing without a working network connection, the application has to be able to run in offline mode. This is achieved by recording all sensor data (both standard variables and high frequency data) on the device’s SD card. Note that even if a network connection is available, data is still stored locally, in case a problem arises on the server. If an SD card is not present and writeable, the device’s internal storage is used instead. With more recent versions of Android (4.0 and above), this distinction is not necessary, as the internal storage and SD card are treated as one contiguous storage space. Tracks that are recorded without a working network connection are designated as ‘offline tracks’, and are uploaded to the server at the earliest opportunity, as described in section 3.2.5.

Tracks are stored on the SD card in the following structure:

```
<device_storage_directory>/skitracker/<users_name>/<track_name>/
```

Skitracker records two types of data, which are handled differently:

1. Variables such as GPS position, speed, distance and any physiological parameters, are stored together as points in a file that represents a track. The interval between these points can be set by the user, and is typically one or two seconds. These points, and the track file as a whole, adhere to the standard of the Track API, meaning that data can be sent to the server in real-time and immediately viewed using this API, without any conversion or further computation.

2. High frequency data, for example 100Hz readings from an accelerometer, are separately recorded in log files and are uploaded to the server when a track is complete. Whilst the Track API does support the collection of this kind of data, transmitting it to the server in real-time is not feasible due to the file sizes involved.

Communication with the server

The application communicates with the web-based system using the Track API, which is based on HTTP requests, and all data that is recorded on the device is uploaded to the server either in real-time, or at a later time if an internet connection is not available at the time of recording.

When running with an active internet connection, the Skitracker system is a very powerful instant feedback and analysis tool, using the mobile device to upload data in real-time, and the web interface to view this data very soon after. In fact, when the device is configured by the user to send data every second, the end-to-end time (from device to server and back to another user’s web browser) is typically around three seconds. The downside, however, is that this whole process is very susceptible to the problems that HTTP requests normally face, namely varying latency and packet loss. Typically the
application will be connected to the internet via GPRS or HSDPA when the skier is
skiing, which further exacerbates these problems.

Therefore, when designing the application (and the Track API to some extent), we
have implemented methods to ensure that all the track points are received at the server,
intact and uncorrupted. This means that the tracks stored on the device’s SD card and
on the server are identical once a track has been completed. From time to time, HTTP
requests sent from the device may timeout or fail to reach the server for some reason.
This is an unavoidable problem, especially when using mobile networks, as HTTP makes
no guarantee of request delivery. Therefore we have implemented our own ACK system,
whereby requests are re-sent if they timeout. This is outlined below:

1. When a data point (in the form of a JSON object) needs to be sent to the server, it
is placed in a buffer. Each point is indexed in this buffer by its timestamp, to avoid
duplicate points being sent. The nominal length of this buffer can be set by the user
in the application’s settings. This enables data points to be sent in groups, which
reduces network traffic.

2. Once the buffer is full, it is time to send the its contents to the server using the
Track API’s update command. The device uses an HTTP POST request to sent a
JSON array that represents the data points in the buffer. No data is removed from
the buffer at this stage.

3. When the server receives the data points and has processed them successfully, it
returns an ACK that contains the timestamps of each point it received.

4. Upon reception of these timestamps the buffer is searched, and any points that have
a timestamp found in the ACK are removed from the buffer. Any data points for
which an ACK was not received remain in the buffer and will be re-sent when the
buffer is full again.

Note that the buffer’s size is not a hard limit: it may sometimes contain more points
than the specified number. For example, this may occur if a group of five points was sent
but no ACKs were received, and a new data point is added to the buffer before another
transmission attempt has been made. In this case, the buffer would temporarily contain
six points until the next set of ACKs is returned.

This buffer system does not wait a specified time before resending a point, rather
it resends the point with the next available request. This does lead to points being
retransmitted several times, which is not as efficient as if the system were to wait for a few
seconds before resending a point. However, our primary concern is that the data reaches
the server as fast as possible, because a user may be monitoring the data in real-time. The
Track API handles duplicate points as standard, so this approach does not cause problems
with the track data.

The issues of out-of-order HTTP requests and corrupted data points are covered in
the description of the Track API, as these problems are handled at the server.

Syncing offline tracks

Once an offline track has been completed, its name is stored on the device to indicate
that it needs to be uploaded (synced) to the server. When the application detects that a
network connection is available the entire track, including all log files, is uploaded to the
server using the addTrackFile and addLogFile Track API commands. As the files involved
may be large (a two hour sensor log file containing 50Hz accelerometer data is in the order of 30MB in size), the upload is compressed using the Deflate lossless compression algorithm [2]. In the case of the sensor log files, which are pure text, a compression ratio of typically 10:1 is achieved.

This syncing mechanism is capable of handling multiple tracks, so the user can record several offline tracks in a session before a network connection is restored. The uploading of tracks happens automatically in the background and requires no user interaction, although a status bar notification is generated to alert the user, which is standard behaviour for Android applications.

If for some reason a track fails to sync with the server properly, it is saved and included in the next sync. Additionally, the user may upload the track to the server manually at any time, by going to the My Tracks tab and uploading it from there.
3.2.6 Multithreading and periodic execution

Any application that uses HTTP (and the Internet as a whole) for data exchange will experience the issue of latency, and the subsequent blocking of execution, that is inherent with network interaction. This issue is exacerbated when using wireless networks, so our application has to be delay-tolerant without affecting the user experience. The well-established solution to this is to use separate threads for the user interface and for any blocking tasks. However, there are several ways to achieve this.

The normal method in Java is to use a shared Handler object that threads can use to send messages to each other. However, Android provides a simpler, smarter way to manage this: the AsyncTask. This is a wrapper class included with the Android SDK that allows blocking code to be run on a background thread, but also allows actions to be performed on the main UI thread before, during and after execution. Underneath, this class is still using threads and a Handler, but it is presented to the programmer in a simpler way, required fewer lines of code to achieve the same result. Memory management also benefits from using this approach, as when an AsyncTask has finished executing, it may be reused by the OS’s thread management system. The following code is taken from our application, and uses an AsyncTask to connect to a remote server using HTTP. Some none-relevant lines have been removed for clarity.

```java
/* We extend the AsyncTask class, and the three parameters are the parameter inputs of each of the methods in the class*/

class ConnectToServer extends AsyncTask<Void, Void, Boolean> {
    // Get the user parameters and add them to the request
    String response = ""; // HTTP response data goes in here

    // onPreExecute() is called by the OS first, on the UI thread
    protected void onPreExecute(){
        // Do some stuff on the UI thread here, e.g. launch a dialog
    }

    // doInBackground() is called when onPreExecute() has finished, on the background thread*
    protected Boolean doInBackground(Void... arg0){
        // This is the HTTP request that blocks for some period of time
        response = HTTP.HTTPGET(someURL);
        // Check the HTTP response code and do something received data
        if(HTTP.parseErrors(response)==null){
            // Do something with the response
            return true;
        }
        else return false; // Connection not successful
    }

    // onPostExecute() is called when doInBackground() has finished on the UI thread*
    protected void onPostExecute(Boolean success){
        if(success) trackStarted(); // Run some code on the UI thread
        return;
    }
}
```

Listing 3.1: Multithread using Asynctask
Our application uses AsyncTasks for any task that involves long running blocking code, but still needs to interact with the user in some way. The vast majority of such cases involve HTTP communication. For blocking code that does not require user interaction, we use a basic thread, as we do not require access to the user interface via the onPreExecute() and onPostExecute() methods. A good example of this is when the application initiates a connection to an external Bluetooth sensor node.

To record sensor data on the device and transmit it at regular intervals, the application needs to run a form of periodic task, where the period of the task is the rate at which the internal and external sensors connected to the system are sampled. The simplest way to achieve this is to run a while loop with a delay command in each iteration. In Java, a delay can be implemented using the Thread.sleep(time_in_ms) method. However, because the application needs to always be able to respond immediately to user actions, it is not acceptable to call this sleep method from the UI thread. If we were to do this, the application’s UI would be unresponsive for the duration of the sleep period. For this reason, we again use an AsyncTask for this, and place the whole while loop, including the code for the update itself and the sleep command, inside the doInBackground() method. This allows other threads in the application (and in other applications running on the device) to execute without being blocked.

Because these periodic tasks may also need to interact with the user, for example, the UI needs to be refreshed when there is new sensor data available, we use another method available to use in the AsyncTask class: publishProgress(). This method can be run at any time whilst doInBackground() is executing, and is run on the UI thread. Its usage is outlined below.

```java
protected Boolean doInBackground ( Void ... arg0 ) {
    // In here we have a loop that executes whilst a status variable is true
    while ( running ) {
        doSomePeriodicAction (); // e.g. save sensor data to the SD card
        // Calls onProgressUpdate() on the UI thread
        publishProgress ( some_data );
        Thread.sleep ( interval );
    }
}
```

```
// onProgressUpdate() is called by the OS when publishProgress() is called
protected Void onProgressUpdate ( some_data ) {
    doOperationsOnUserInterface (); // e.g. update sensor readings on screen
}
```

```
// Call this method to stop the loop after the current iteration
protected void stopRunning () {
    running = false;
}
```

Listing 3.2: Implementing a periodic task using AsyncTask

In the above example, the background thread does all the heavy lifting, whilst also triggering UI updates on the main thread. This approach is both thread-safe and resource efficient.

One final issue with periodic tasks that must be tackled is that of drift. Drift occurs when a fixed delay interval is set in a periodic task, but the execution time of the code in the task is not taken into account. For example, in the above example, if saving the
data to the SD card takes 20ms, and the sleep duration is fixed at 1000ms, then the total period for one iteration would be 1020ms, meaning that sensor data is not being sampled at exactly 1000ms intervals. This is not desirable from a data analysis point of view, so it must be avoided. The solution is to time how long the execution of the code takes, and subtract this from the delay time.

```java
int interval = 1000;  // Desired interval is one second

while (running) {
    long startTime = System.currentTimeMillis();
    doSomePeriodicAction();  // e.g. save sensor data to the SD card
    publishProgress(some_data);
    long endTime = System.currentTimeMillis();
    long executionDuration = endTime - startTime;

    // Subtract execution time from the interval and sleep for this time
    Thread.sleep(interval - executionDuration);
}
```

Listing 3.3: Eliminating drift from a periodic task

Now, if saving the data to the SD card takes 20ms, then the delay will be reduced to 980ms to compensate for this, thereby preserving the overall period of the iteration and eliminating drift. This also works well if the periodic action takes varying lengths of time to execute, for example an HTTP request to a remote server.
3.2.7 User Interface

Introduction

When developing the user interface for the Android application, the focus was to enable the user to easily connect to various sensors, view maps, graphs and data in real time, monitor remote skiers and their data, and configure track settings. Therefore during design we paid attention to the Android user interface guidelines, which explain how to accommodate multiple screen sizes, densities and orientation, and how to optimise the layout for performance.

The layouts of the various screens are defined using XML, with visual elements represented in a set of nested tags, similar to HTML. These XML files are rendered, or 'inflated' by the operating system at run-time. The Android developer toolkit provides a tool for analysing the performance of these layouts, which measures the time it takes for the operating system to measure and render each on-screen element, thereby allowing the designer to see which parts should be optimised.
Activity feed

Built into the home tab, the Activity feed allows the user to see any recent and ongoing activity from other skiers. Touching an ongoing (live) entry will take the to the remote monitoring tab, which is described later.

![Activity Feed Image]

Figure 3.2: The livefeed content has one active skier to be monitored.

29
My Tracks

In this second tab, a list of all tracks the current user has recorded on the device is displayed. By touching one of the tracks, the user can rename, delete and upload the track to the server, and also view graphs of the track’s data (if the track exists on the server).

![Image of My Tracks tab with selected track]

Figure 3.3: User has selected a track so a dialog is displayed to either rename, delete and upload the track
The ’Record track’ tab

This is the tab where new tracks are recorded. Start, stop and lock controls are provided at the bottom of the screen, along with a button that toggles the display of a live map. Data received from both internal and external sensors is shown in real-time in a set of seven boxes, the contents of which may be changed simply by touching the box and choosing the data to display from a list. The six smaller boxes show only a value, whereas the larger top box can either show a value, or a moving bar that represents the value in a predefined range.

Figure 3.4: Here a track is running, with data received from physiological sensors and internal sensors displayed in the seven boxes. The larger box is showing the skier’s heart rate with respect to the user-defined minimum and maximum values.

This is useful for visualising heart rate with respect to some desired minimum and maximum values, for example. These values may be set by the user by touching the ’Set
values’ button. The mode of the larger box may be changed using the orange arrows.

Touching the ‘Map’ button will replace the six smaller boxes with a map, showing the position of the skier and their current track. The transition between the boxes and the map is animated to help the user understand how the application works.

Figure 3.5: When the ‘Map’ button is touched, the display changes to show a live map.
Problem detection

When a user wants to start a track but Skitracker detects a problem, the application will display a message, and show a button which will take the user to the relevant settings screen so they may fix the problem.

![Problems detected]

Skitracker has detected the following problems that may affect tracking. You can fix them using the buttons below, or ignore them and start the track.

- **No internet connection is available**
  - Network settings

- **Bluetooth is not switched on**
  - Bluetooth settings

![Back and Start track buttons]

Figure 3.6: The user has attempted to start a track, but Skitracker has detected that there is no internet connection or Bluetooth connection. The user may touch 'Network settings' or 'Bluetooth settings' to fix these issues, or ignore them and start the track.
Figure 3.7: When a track is finished, a concise summary is shown to the user. Note the 'Upload data' button, which allows the user to upload additional log files to the server.
Adding sensors

For receiving data from external sensors user needs to add Bluetooth connection between external sensors and mobile phone and this can be done directly from Skitracker. Skitracker will scan for both internal and Bluetooth sensors and display any sensors found in a dialog box. Touching on a sensor will show a description of the sensor and connect it to Skitracker. Note that if the sensor is a Bluetooth sensor and has not been paired to the device before, it will need to be paired at this point, a process which is handled within Skitracker.

Figure 3.8: A sensor scan is run in the background and displays any discovered devices. Here, a Zephyr Bioharness has been discovered.
Figure 3.9: In the 'Sensors' screen, a list of sensors currently connected to the system is displayed. Here we see three sensors (two internal on the device and one Bluetooth) are connected and ready to send data to Skitracker.
3.3 The Track API

3.3.1 Introduction

Skitracker is a fully-integrated system where sensor data can be recorded on multiple mobile devices simultaneously in real-time. Additionally, data from these devices can be viewed live on any platform. These two requirements mean that there is a great deal of communication (both static and real-time) between the mobile devices, the host server and any other connected users.

Early in the development stage it became apparent that a custom communication protocol was needed to handle both the transmission of sensor data to the server and the visual presentation of this data, and so the ‘Track’ API was born. HTTP was chosen as the underlying transport mechanism for the API, as it is well established on all platforms. Each command in the API consists of a single GET or POST request.

The Track API is a multi-layered set of commands that can be used to transmit sensor data from one device to another, retrieve time series of sensor data and present these series in maps and graphs. In our system, these time series are referred to as ‘tracks’. To use the API, all that is required is a web server running PHP and MySQL. In order to send data to the server, a device simply needs to be able to perform HTTP requests, and to view maps and graphs a web browser (or embedding rendering engine) is necessary.

HTTP was chosen as the communication protocol because it is supported across all mobile devices. Whilst it does introduce some overhead, we believe it’s simplicity and pervasiveness make it the best choice for the Track API.

The API has been designed to be easily used in any application where the transmission and storage of data in time-series is required. It is easily extensible, requiring only one line of code to be added in a configuration file in order to add a new variable to the system, and has already been used successfully in other projects.

3.3.2 Architecture

The Track API consists of three layers, these are:

1. The accounts layer
   This is a MySQL database which contains details of all users registered with the system, and a set of PHP functions. Users can create accounts and login / logout using their user ID and password. Each user has an API key associated with their account, which is used for all API access, and provides a security mechanism, allowing users to control who can view their recorded data.

2. The data layer
   This is the intermediate level of the system, and can be used to get raw data from a track, upload data to a track and to edit track the track’s properties. The data output can be customised in a number of ways, which will be detailed later. This layer can be accessed from any platform that supports HTTP requests.

3. The visual layer
   This layer is used to take data from the data layer and present it in graphical format. This can mean plotting a track on a map, or graphing any number of variables against time or against each other. This layer can be used only in a web browser or embedding rendering engine (such as Android’s WebView).
3.3.3 Lifecycle

During its lifetime, a track is created, updated zero or more times, then archived. Data may be retrieved from the track when it is either running (i.e. realtime) or when it is archived (a past track). Once complete, a track may be edited using commands built into the API. The figure below illustrates the lifecycle of a typical track.

Figure 3.10: The track lifecycle. The two possible track states are shown in green, whilst commands are shown in blue.

A complete guide explaining how to use the track API can be found in the appendix, section 6.1.
3.4 Web interface

3.4.1 Introduction

Skitracker provides a fully-featured web interface for viewing data from real-time and archived ski tracks. This interface allows skiers to view maps and graphs of data recorded during a track, such as data recorded by the device’s GPS, heart rate from a physiological sensor belt, or skiing technique data recorded using accelerometers. These maps and graphs used are generated using the aforementioned Track API.

Additionally, the user may perform other tasks on tracks they have recorded, for example setting a track’s lap points, running technique analysis on the track, or exporting a subset of the track’s raw data for external analysis.

The web UI itself is generated using PHP and MySQL, and presented using HTML and Javascript, meaning it can be used in any browser or on any device that adheres to W3C standards.

3.4.2 Features

Viewing data

The web UI is designed to offer something to all kinds of users. For the casual skier, a news feed detailing recent activity is shown on the home page. This feed is dynamic, and will update its contents when a skier is recording a track. Clicking on an item in the feed will take the user to a page where various track data can be seen using a set of easy to use maps and graphs. Professional skiers and their coaches may customise and filter the data displayed according to their needs, making the UI a powerful tool for performance analysis.

Maps

If the track selected contains GPS data, then the user can view a map that shows the track, as shown in the figure below. It should be noted that if the track is a live one, the map will periodically update itself with new GPS data. Lap points are represented on the map by markers, and are labelled for easy identification.

The map may be optionally viewed together with a single line graph of any variable underneath it, where hovering with the mouse over a graph or map point will highlight the corresponding map or graph point, respectively. This allows the user to relate a point in time on the graph to a position on the track.

A zoom feature is provided as part of the interface, which enables the user to view any subsection of the track, simply by clicking on the first and last points of the section they wish to view on either the map or the graph. Any number of sequential zoom operations may be performed.
Figure 3.11: A map of a track with an accompanying graph of heart rate. In this example, a point on the graph has been hovered over, causing the large marker on the map to be displayed at the corresponding map point. Note the options at the top, where the variable to be graphed is selected, along with the filter type and size.
Line graphs

If the user wishes to see how other variables such as heart rate or speed change over the course of the track, they can view data as line graphs. Multiple variables may be viewed at the same time, one above the other, so that the relationship between them can be seen easily. To aid the user, when the mouse is placed over a point in one graph, the corresponding points in the other graphs are highlighted. As with the maps, graphs of live data are updated in real-time, and zooming is supported.

Figure 3.12: A set of three line graphs showing the speed, elevation and heart rate of a skier during two laps of a race. The laps are clearly defined, with features in each graph being repeated. The selected point on the graphs is the start of a steep downhill section of the course, which is characterised by a rapid increase in speed, coupled with a decrease in heart rate.
Scatter graphs

Scatter graphs are also available to the user, allowing them to easily plot one variable against another in order to determine if there is a correlation between them or not.

Figure 3.13: A scatter graph of body posture plotted against speed. Two distinct clusters can be seen, centered around 25 and minus 20 degrees. Hovering on a point displays the co-ordinate values of the point.
Track statistics

Using the web interface, it is also possible to see a statistical breakdown of a set of variables, which is useful when raw data is required instead of a graph. In this view, minimum, average and maximum values for each selected variable are displayed, with the data split into sections denoted by the track’s lap points, if any have been set.

Figure 3.14: A demonstration of the statistics viewer. Here we see eight one kilometre sections of a track.

Extending this concept further, the web UI also provides a statistical breakdown of skiing technique information, if it is present in the track. By default, when an accelerometer log file (either from the device or a sensor belt) is uploaded by the user, technique classification is run on it. This classification is executed by an external program, which is triggered by a PHP script on the server, and provided with a set of reference files, and the accelerometer log file itself. The program then compares the log file with the reference files and produces several output files that contain the results of its analysis, including gear numbers, motion amplitude, symmetry and consistency for each period that was successfully classified. This data is subsequently merged with the existing track file.

In this view, detailed information about which gears were used is displayed for each section, including the time spent in each gear, the number of cycles of each gear, and other performance indicators such as motion amplitude, consistency and symmetry. The left and right forms of asymmetric gears, namely gears two and four, are shown separately, as these need to be analysed independently.
Figure 3.15: The gear statistics view. From the data, it is clear that the skier in question prefers to use the right form of gears two and four.
Managing and editing tracks

The Skitracker system provides the user with a powerful set of editing and management features, grouped together into a page called the Track Control Panel, which can be used to:

- Set a track’s lap points.
  
  Lap points are used to break a track down into sections, thus allowing different sections of a track to be compared. In the normal case, the lap points are set by the device at regular distance intervals. However, these points may be changed after a track has been completed. To do this, the user is provided with a map of the track (generated using the Track API), and may add lap points by simply clicking on the line that represents the track. Points may also be removed by clicking on them.

  The user also has the option of running automatic lap detection, whereby the system will attempt to find lap points by looking to see if the skier went past the same point multiple times during the track. This makes use of the Track API command `runLapDetection`, which is described in the appendix, section 6.1.4.

- Run technique analysis.
  
  As mentioned earlier, technique classification is automatically run when a log file containing suitable accelerometer data is uploaded. This automatic classification is run against a general set of references that is designed to match the vast majority of skier’s techniques. However, each user may have their own technique profiles in the system, based on data recorded from their own skiing, which will provide more accurate classification as the reference files are ‘tuned’ to their particular technique.

  In the track control panel, an interface is provided that allows the user to choose which profile to classify their data against. Additionally, if multiple suitable accelerometer log files have been added to the track, then the user may choose which log file to use as input.

- Export track data.
  
  The user may export any set of variables from a given track, in either `csv`, `json` or `xml` format. For example, from a track containing GPS and physiological variables, the fields ‘distance’ and ‘speed’ may be exported on their own, ignoring any other fields. The user not only has the choice of which format to export the data in, but also whether to download the data as a file, or simply have it output ‘as is’ on the page, ready for copying into another application.

- Restore a track.
  
  When a track is completed, a backup copy is created and saved on the server. This is built in to the operation of the Track API. If at any time during editing the user wishes to undo any changes made to the track, they may restore it from the backup file.

  It should be noted that all of the above management operations are performed using commands in the Track API, and may only be performed by the owner of the track. The system is designed in this way to ensure that the only way a track may be read from or written to is via the API, which both increases security and decreases the chance of something going wrong.
3.5 Shimmer sensor nodes

3.5.1 Introduction

Shimmer research group is the leading provider for wearable sensor nodes they design, develop and manufactures wearable wireless sensors including Kinematic, Bio-Physical and Ambient modules. Skitracker system benefits from one of their products which is Shimmer 2 sensor node, which has been available since 2009. It’s a small and light sensor platform, suitable for wearable applications with a low power consumption. This node has integrated sensors accelerometer, gyroscope and for communication IEEE 802.15.4 Radio and Bluetooth Technology, one another feature is that it handles local storage in a microSD card. The shimmer could be connected to external sensors by using an extension board, therefore the shimmer node could be defined as a wearable, robust, reliable node for capturing parameters and communication in real-time to devices with Bluetooth technology integrated with programmable leds on the board. The shimmer research group is supporting the idea of open source, which makes shimmer node an efficient and effective wireless sensing firmware. To mention that shimmer node fully supports TinyOS, this operating environment is highly recommended because of its supportive of platforms open-source code library and targeting wireless-sensor networks platform.

3.5.2 TinyOS and nesC programming language

While designing the system we decided to install TinyOS 2.1.1, which is free and suitable for our purpose to create tiny and optimised programs of wireless sensor networks. The TinyOS is an embedded operating system and supports nesC (network embedded systems C) program based applications which is built as an extension of C-programming language but supports the TinyOS concurrency model which is a component based, event-driven programming language. Shortly the nesC language supports a model that integrates reactivity to the environment, concurrency, communication and data race detection. The importance of using nesC as the programming language is that it provides an optimised program and simplifies application development, reduces code size and eliminates many sources of potential bugs. [12]

- Component-based architecture: Composed of multiple components wired to form an executable application and there are two types of components, module and configurations. The module handles the application code and configurations are used to wire the components together.

- Tasks and event-based concurrency: Tasks don’t preempt but may deferred computation while events may preempt but runs to completion.

- Split-phase operations: The codes return immediately and completion of this will be signaled by an event.

A component provides and uses interfaces these interfaces are the only point of access to the component and are bi-directional. Interface declares a set of functions called commands that the interface provider must implement and another set of functions called events that the interface user must implement. One another important feature nesC is handling is the atomic statement, which is basically used in the application where we want it to execute without any preemption.
3.5.3 Design and Implementation

The shimmer node allows the system to receive pole force data by connecting its ADC pin to LSB200 sensor via an extension board, various interfaces have been declared to provide this feature of shimmer. The program provides configuration wires the components together and modules initialise the access to the interfaces and handles the logic execution of the application. To have a fully-featured program to read from the ADC pin and stream these data over Bluetooth to the mobile device periodically each 100ms there are a few important interfaces that have to be declared. Shimmer node has to be booted with initialisations of hardware components to access these features such as Bluetooth and accelerometer sensing. The tiny program loaded to the node executes according to nesC language’s architecture, the task and event executes and an asynchronous event read from the ADC channel to access the sensor data via the Shimmer’s DMA. The data is saved into a buffer to be added into a package to be sent over Bluetooth to a mobile device. Figure 3.16 shows the communication between shimmer node and sensor, and figure ?? shows how how interfaces are used to connect components together.

Figure 3.16: Graphical overview of shimmer nodes
The package format of the sensor data to be sent:

\[//X1023Y1023Z1023P1023\] (3.1)

The number after each letter defines its sensing data:

- **X**: x-axis accelerometer data
- **Y**: y-axis accelerometer data
- **Z**: z-axis accelerometer data
- **P**: pole for data

Note: The mobile device receives this package and after splitting the package to individual data we calculate peak force, time of peak and cycle in the mobile device. We could had calculated these measurements in the shimmer node but because of its performance and memory of the mobile device it was more effective to handle these operations in the mobile device.
3.6 System overview

Figure 3.17: An overview of the Skitracker system. Data from various wireless sensors is collected by an Android smartphone using Bluetooth. This data is then processed and displayed on screen for the user to see. This data is additionally transmitted to a remote server using the Track API. Once the data is stored on this server, it may be viewed and edited using our web interface, which also uses the Track API to retrieve data.
Chapter 4

Testing
4.1 Testing the system

The Swedish cross-country ski team 2015 had a training camp in Toblach, Northern Italy in November 2012, where we had the chance to test our system on male and female elite-level skiers. The mobile device to detect various techniques (gears), BioHarness physiological belt to detect techniques and physiological data (gears, heart-rate, breathing-rate) and Zephyr shirt to detect techniques and physiological data (gears, heart-rate). Which ever equipment the skier preferred to use they received parameters the equipment provides but all of them had a mobile device to receive GPS signal and also being the bridge to collect data. However the GPS signal during some training days and location wasn’t as we hoped for, the weather condition and location affected the GPS signal and also quality of GPS on the mobile device could had been better to receive better result. To understand the physiological data properly more deep data analysing could be done, the estimated maximum heart-rate depends on age and how physical strong the person is, so there is a different between athletes and non-athletes. There are lots of discussions and assumptions about athletes having lower maximum heart-rate compare to non-athletes at the same age and how to predict the maximum heart rate. One of the assumptions is that an athlete has a maximum heart-rate of 3-10 minus the maximum heart-rate of a non-athlete at the same age however a few more formulas predict estimated heart-rate for athletes and this is because athletes have hearts that can pump more blood with each beat than the hearts of a non-athlete person, so they do not beat as often [13].

\[ \text{Figure 4.1: Assumption for non-athlete} \]
\[ \text{Estimate-MHR} = 220 - \text{Age} \] \[16\]

\[ \text{Figure 4.2: Assumption for male athlete} \]
\[ \text{Estimate-MHR} = 202 - (0.55 \times \text{age}) \] \[13\]

\[ \text{Figure 4.3: Assumption for female athlete} \]
\[ \text{Estimate-MHR} = 216 - (1.09 \times \text{age}) \] \[13\]

The breathing-rate as heart-rate also changes for athletes and especially for cross-country skiers where they engaged about all the major muscle groups in the body so they have bigger lung capacity and have high maximal oxygen uptake (VO2 max). The intake and uptake oxygen per time is higher therefore their breathing rate per minute is lower than non-athletes. A lot of researches have been done for cross-country skiers which gives a relationship between oxygen consumption and heart-rate for measuring intensity during a training of a skier, which the result corresponds with relative percentage of VO2 max [3]. The first training session we tested our equipments was on a skate training from Cortina d’Ampezzo to Tre Cime di Lavaredo at 9.00 am, here we tested our system on 5 male skiers and 5 female skiers their training was approximately 26.66 km for male skiers, the female skiers had a slightly less distance but it took approximately 2-2.30 hours for the skiers to reach the top. The skiers started from a point of approximately 1200 m height and the finish position was around 2400 m, this means they went around 1200 m uphill and as the elevation increases the body reacts differently which is interesting if you want to analyse the physiological data. When we look at the data by using our tools, we see that the GPS signal was very accurate on most of the skiers which was an average of around 10 m.
The equipments seem to have worked smoothly except on one of the male skier where his equipments lost Bluetooth connection at the very beginning and the only data the system received was GPS data and one of the male skier’s mobile device had low GPS signal. Looking at the result data from this training of one of the male skier’s: Time: 02:16:20, Distance 23.06km, Average GPS accuracy: 11.65 m, Average breathing-rate: 34.92 bpm, Average breathing wave amplitude: 0.14 , Average heart-rate: 136.15 bpm.

### Table 4.1: Equipments used during Tre Cime di Lavaredo training session

<table>
<thead>
<tr>
<th>Devices</th>
<th>Male Skiers</th>
<th>Female skiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioHarness belt</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Zephyr shirt</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mobile device</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The second testing session was on the same day as the Tre Cime di Lavaredo but in the afternoon at 3.00 pm, starting in Italy crossing the border to Austria. On this session we tested our system on 4 male skiers, the test went very good but except that one of the skier has no GPS data and most probably this depends on that he had put the mobile device in his bag where the mobile device couldn’t get GPS signal. The result of one of the skiers during double poling: Time: 01.54.46, Distance: 22.20 Km, Average speed: 11.65 km/h, Average GPS accuracy: 7.97 m, Average breathing rate: 39.06 bpm, Average breathing wave amplitude: 0.11, Average Heart-rate: 121.39 bpm.

### Table 4.2: Equipments used during Tre Cime di Lavaredo training session

<table>
<thead>
<tr>
<th>Devices</th>
<th>Male Skiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioHarness belt</td>
<td>3</td>
</tr>
<tr>
<td>Zephyr shirt</td>
<td>0</td>
</tr>
<tr>
<td>Mobile device</td>
<td>1</td>
</tr>
</tbody>
</table>

The third training session was the second day in Toblach started at 9.00 am, it was a skate race around a track with 3-4 laps where we tested our equipments on 5 male and 6 female skiers.

### Table 4.3: Equipments used during Tre Cime di Lavaredo training session

<table>
<thead>
<tr>
<th>Devices</th>
<th>Male Skiers</th>
<th>Female Skiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BioHarness belt</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Zephyr shirt</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Mobile device</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
4.2 Data analysis

The skate race was held on the last day of the testing days in Toblach, where they skied along a specific track with different elevation, uphill, downhill and flat surface which was ideal to test our system to detect various techniques and to analyse physiological data in different elevations. However as the testing were with skates and not on real snow condition there might have been some differences but we believe the results were closer to real snow conditions compare to the indoor testing on a treadmill. During this skate race the skiers wore the different equipments our system support; the mobile device to detect various techniques (gears), BioHarness physiological belt to detect techniques and physiological data (gears, heart-rate, breathing-rate) and Zephyr shirt to detect techniques and physiological data (gears, heart-rate). Which ever equipment the skier preferred to use they received parameters the equipment provides but all of them had a mobile device to receive GPS, but because of the quality of GPS signal on the mobile device, location and bad weather condition on the skate race day some of the skiers lack a good GPS signal. This was actually one of the most important data to measure the performance however by looking at one of the skier’s good GPS data and the skier’s techniques we could approximately indicate their position and measure their performance.

This skate race was held in two sessions, starting with the women’s race at around 9.20 a.m. they skied three laps of the track which was 12 Km once they had finished the men’s race started but they skied four laps which was 16 Km. The aim of this data analysing is to understand the skier’s techniques during an outdoor training session and their physiological data to have a better understanding of their capacity. How we did this was to look at different elevation of the track comparing the laps to each other and also the skiers to each other. What we understood from this was different skiers might be stronger using different techniques which they might be aware of or maybe not and
different skiers might also be stronger at different uphill. Some skiers might be stronger
during a steep uphill while others not. We hope this data analysing could help the skiers
to have theoretical understanding of their skiing techniques to increase their performance.
For a better data analysing the track is divided into five sections:

**First section:** from start point to first uphill

**Second section:** first uphill

**Third section:** after first uphill to second uphill

**Fourth section:** second uphill and third uphill

**Fifth section:** after third uphill to the finish point

---

Figure 4.5: Skate race map, Toblach

Figure 4.6: Elevation from mobile device GPS, Toblach
4.2.1 Women’s skate race

The first part of our data analysing is from the women’s skate race 12 Km, here we analysed data from two skiers skier1 and skier2. Both of the skiers were wearing the BioHarness physiological belt because of this we could analyse more data, the skier2 had average accuracy of 5 metre GPS signal however skier1 had no GPS signal but we could figure out the position from the techniques at different timestamp. Skier1 was faster than skier2 by 8 seconds and from our analysing we can see that their approached different.

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier1 Time (minutes)</th>
<th>Skier2 Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>10:44</td>
<td>11:02</td>
</tr>
<tr>
<td>Lap2</td>
<td>11:00</td>
<td>11.18</td>
</tr>
<tr>
<td>Lap3</td>
<td>11:11</td>
<td>10:43</td>
</tr>
<tr>
<td>Total time</td>
<td>32.55</td>
<td>33:03</td>
</tr>
</tbody>
</table>

Table 4.5: Average physiological data

<table>
<thead>
<tr>
<th>Data</th>
<th>Skier1</th>
<th>Skier2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart-rate (bpm)</td>
<td>187.07</td>
<td>154.91</td>
</tr>
<tr>
<td>Breathing-rate (bpm)</td>
<td>42.03</td>
<td>39.02</td>
</tr>
<tr>
<td>Breathing wave amplitude()</td>
<td>0.08</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Table 4.6: Maximum physiological data

<table>
<thead>
<tr>
<th>Data</th>
<th>Skier1</th>
<th>Skier2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart-rate (bpm)</td>
<td>238.80(-28)</td>
<td>198.20</td>
</tr>
<tr>
<td>Breathing-rate (bpm)</td>
<td>58.90</td>
<td>54.84</td>
</tr>
<tr>
<td>Breathing wave amplitude()</td>
<td>2.21</td>
<td>0.34</td>
</tr>
</tbody>
</table>

The physiological data are accurate however it seems that skier1’s physiological belt was perhaps not correctly fitted, therefore the we could say maximum heart-rate(-28), and from the estimating maximum heart-rate formula it seems that both of the skiers had been on their maximum. Their breathing-rate is quiet close, but with a different of 4.06 bpm.

Table 4.7: First Section: 180m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier1 Speed (m/s)</th>
<th>Skier2 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>2.86</td>
<td>5.45</td>
</tr>
<tr>
<td>Lap2</td>
<td>2.86</td>
<td>4.29</td>
</tr>
<tr>
<td>Lap3</td>
<td>2.86</td>
<td>5.29</td>
</tr>
</tbody>
</table>

*Skier2 had a higher speed on this section*
Table 4.8: Second Section: 590m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier1 Speed (m/s)</th>
<th>Skier2 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>5.22</td>
<td>4.72</td>
</tr>
<tr>
<td>Lap2</td>
<td>4.61</td>
<td>5.00</td>
</tr>
<tr>
<td>Lap3</td>
<td>4.57</td>
<td>5.04</td>
</tr>
</tbody>
</table>

In overall skier2 was faster on this section but in the first lap skier1 had a higher speed.

Table 4.9: Second Section: Techniques Lap1

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier1</td>
<td>43</td>
<td>0</td>
<td>34</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier2</td>
<td>0</td>
<td>66</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.10: Second Section: Techniques Lap2

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier1</td>
<td>68</td>
<td>2</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier2</td>
<td>0</td>
<td>61</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.11: Second Section: Techniques Lap3

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier1</td>
<td>71</td>
<td>1</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier2</td>
<td>2</td>
<td>53</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

On this section we analysed the skiers techniques, the skier1 on all the three laps started with gear 3 then switched to gear 2. She seems to had preferred gear 2L compare to 2R, and her fastest lap was Lap1 here she approached differently where she used gear 3 more compare to the other laps. Here we can ask if she was faster because she used gear 3 more or was it because it was her first lap. Skier2 also started with gear 3 and then switched to gear 2, she seems to prefer gear 2R compare to gear 2L. What we could see is that she used the same techniques and switched the gears almost the same in all the three laps.
Figure 4.7: *Skier1’s physiological data during second section and a small part from section three on Lap1*

Looking at skier1’s heart-rate graph we see how the heart-rate increases and how fast the recovery is, by looking at breathing wave amplitude and breathing-rate we understand her physiological capacity, how much oxygen she intakes each time.
Figure 4.8: Skier2’s physiological data during second section and a small part from section three on Lap1

Looking at skier2’s hear-rate we see how it increases and decreases during the uphill, which is different from skier1 but the recovery seems to very fast and by looking at the breathing wave and breathing amplitude we can see it’s only at maximum at the beginning of the uphill which should be questioned, as she should be able to breath more oxygen each time.

Table 4.12: Third Section: 260m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier1 Speed (m/s)</th>
<th>Skier2 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>20</td>
<td>11.81</td>
</tr>
<tr>
<td>Lap2</td>
<td>15.29</td>
<td>9.29</td>
</tr>
<tr>
<td>Lap3</td>
<td>11.30</td>
<td>10.83</td>
</tr>
</tbody>
</table>

*Skier1 was faster on this section*

Table 4.13: Fourth Section: 330m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier1 Speed (m/s)</th>
<th>Skier2 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>2.24</td>
<td>2.82</td>
</tr>
<tr>
<td>Lap2</td>
<td>2.14</td>
<td>2.77</td>
</tr>
<tr>
<td>Lap3</td>
<td>2.54</td>
<td>3.03</td>
</tr>
</tbody>
</table>

*Skier2 was faster on this section*
Table 4.14: Fourth Section: Techniques Lap1

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier1</td>
<td>60</td>
<td>2</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier2</td>
<td>0</td>
<td>60</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.15: Fourth Section: Techniques Lap2

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier1</td>
<td>61</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier2</td>
<td>11</td>
<td>60</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.16: Fourth Section: Techniques Lap3

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier1</td>
<td>67</td>
<td>4</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier2</td>
<td>0</td>
<td>51</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

On this section, skier1 started with gear 3 and then switched between gear 2 and gear 3. She preferred gear 2L compare to 2R, it’s hard to tell why her speed varies between the laps, she was fastest on Lap3 where she has also switched to gear 2R but otherwise she used approximately the same techniques over this section. Skier2 also started with gear 3 then switched between gear 2 and gear 3. She seems to have preferred gear 2R compare to gear 2L but on Lap2 she has used gear 2L where this lap was her slowest of this section. But of the skiers fastest on Lap3, we could ask if they put on their maximum on this lap.

Table 4.17: Fifth Section

<table>
<thead>
<tr>
<th>Skier1</th>
<th>Skier2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laps</td>
<td>Speed (m/s)</td>
</tr>
<tr>
<td>Lap1</td>
<td>8.75</td>
</tr>
<tr>
<td>Lap2</td>
<td>8.55</td>
</tr>
<tr>
<td>Lap3</td>
<td>8.42</td>
</tr>
</tbody>
</table>

In overall Skier1 was faster on this section

Looking at all the data, skier1 seems to be faster compare to skier2 during flat surface or downhill while skier2 seems to be faster during uphill. They used different techniques which could mean they prefer and are stronger at different gears, skier1 seems to prefer gear 2L while skier2 seems to prefer gear 2R. The physiological data seems to be very accurate, however it’s a bit hard to understand skier2’s physiological data where it the heart-rate increases and decreases rapidly and the breathing wave amplitude seems to not be quiet what we would expect.
4.2.2 Men’s skate race

The second part of our data analysing is from the men’s skate race 16 km, here we analysed data from two skiers skier3 and skier4. Skier3 was wearing the Zephyr physiological shirt and had GPS signal with average accuracy of 12.15 m while skier4 was wearing the mobile device and also had GPS signal with average accuracy of 9.44 m. Here we analysed mostly the techniques and also heart-rate data of skier3. Skier2 was faster than skier1 with approximately 1.45 minutes, from our analysing we understand that they used different techniques and seems to be stronger at different sort of surface.

Table 4.18: Skate Race

<table>
<thead>
<tr>
<th>Laps</th>
<th>Time (minutes) Skier3</th>
<th>Time (minutes) Skier4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>9.28</td>
<td>9.14</td>
</tr>
<tr>
<td>Lap2</td>
<td>9.36</td>
<td>9.19</td>
</tr>
<tr>
<td>Lap3</td>
<td>9.55</td>
<td>9.11</td>
</tr>
<tr>
<td>Lap4</td>
<td>9.35</td>
<td>9.05</td>
</tr>
<tr>
<td>Total time</td>
<td>38.34</td>
<td>36.49</td>
</tr>
</tbody>
</table>

Table 4.19: Average physiological data

<table>
<thead>
<tr>
<th>Data</th>
<th>Skier3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart-rate (bpm)</td>
<td>184.6</td>
</tr>
</tbody>
</table>

Table 4.20: Peak physiological data

<table>
<thead>
<tr>
<th>Data</th>
<th>Skier3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heart-rate (bpm)</td>
<td>196.0</td>
</tr>
</tbody>
</table>

The heart-rate data seems to be very accurate according to the estimated maximum heart-rate formula, which would mean that he had given his maximum during the race.

Figure 4.9: Skier2’s physiological data and elevation during the whole race
Table 4.21: First Section: 180m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier3 Speed (m/s)</th>
<th>Skier4 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>3.05</td>
<td>3.00</td>
</tr>
<tr>
<td>Lap2</td>
<td>2.86</td>
<td>3.53</td>
</tr>
<tr>
<td>Lap3</td>
<td>3.00</td>
<td>2.73</td>
</tr>
<tr>
<td>Lap4</td>
<td>3.10</td>
<td>3.05</td>
</tr>
</tbody>
</table>

*In overall skier2 had higher speed*

Table 4.22: Second Section: 590m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier3 Speed (m/s)</th>
<th>Skier4 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>6.63</td>
<td>7.38</td>
</tr>
<tr>
<td>Lap2</td>
<td>6.08</td>
<td>6.63</td>
</tr>
<tr>
<td>Lap3</td>
<td>5.9</td>
<td>6.94</td>
</tr>
<tr>
<td>Lap4</td>
<td>5.73</td>
<td>6.86</td>
</tr>
</tbody>
</table>

*Skier3 had higher speed*

Table 4.23: Second Section: Techniques Lap1

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>1</td>
<td>14</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier4</td>
<td>0</td>
<td>4</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.24: Second Section: Techniques Lap2

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>4</td>
<td>23</td>
<td>33</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier4</td>
<td>0</td>
<td>12</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.25: Second Section: Techniques Lap3

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>1</td>
<td>51</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier4</td>
<td>0</td>
<td>14</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.26: Second Section: Techniques Lap4

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>5</td>
<td>35</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier4</td>
<td>0</td>
<td>14</td>
<td>39</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
On this section we analyse the skier’s techniques used, skier3 started this section with gear 3 and continued with gear 2 however in Lap2 and Lap4 he seems to had switched between the gears. He definitely preferred gear 2R compare to gear 2L, and his fastest lap was Lap1 here we could ask if this was because it was the first lap or was it because during this lap he preferred gear 3 more compare to the other lap’s of this section. Skier4 different from skier3 started with gear 3 then switched to gear 2 during all the laps however he seems to prefer gear 2R where he didn’t switch to gear 2L at all. Skier3 had higher speed at this section, we could ask if it was because of the sort of uphill where he might be faster during long uphill.

Figure 4.10: Skier2’s physiological data during second section and a small part from section three on Lap1

Looking at this graph we can see the heart-rate against the elevation, which indicates how heart-rate increases while he goes uphill and how fast the recovery is.

Table 4.27: Third Section: 260m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Skier3 Speed (m/s)</th>
<th>Skier4 Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>12.38</td>
<td>10.83</td>
</tr>
<tr>
<td>Lap2</td>
<td>17.33</td>
<td>11.81</td>
</tr>
<tr>
<td>Lap3</td>
<td>15.29</td>
<td>14.40</td>
</tr>
<tr>
<td>Lap4</td>
<td>21.66</td>
<td>14.4</td>
</tr>
</tbody>
</table>

*Skier3 was faster on this section.*
Table 4.28: Fourth Section: 330m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Speed (m/s)</th>
<th>Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>3.47</td>
<td>4.02</td>
</tr>
<tr>
<td>Lap2</td>
<td>3.17</td>
<td>3.51</td>
</tr>
<tr>
<td>Lap3</td>
<td>3.20</td>
<td>3.84</td>
</tr>
<tr>
<td>Lap4</td>
<td>2.75</td>
<td>3.66</td>
</tr>
</tbody>
</table>

*Skier4 was faster on this section*

Table 4.29: Fourth Section: Techniques Lap1

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>4</td>
<td>46</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Skier4</td>
<td>0</td>
<td>24</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.30: Fourth Section: Techniques Lap2

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>9</td>
<td>45</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Skier4</td>
<td>0</td>
<td>35</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.31: Fourth Section: Techniques Lap3

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>4</td>
<td>45</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Skier4</td>
<td>1</td>
<td>28</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.32: Fourth Section: Techniques Lap4

<table>
<thead>
<tr>
<th>Gears</th>
<th>2L</th>
<th>2R</th>
<th>3</th>
<th>4L</th>
<th>4R</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skier3</td>
<td>4</td>
<td>45</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Skier4</td>
<td>2</td>
<td>29</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

On this section, skier3 started with gear 2L and then switched between gear 2 and gear 3. He seems to have preferred gear 2R compare to 2L however almost at the same point he has switched to gear 2L during the four laps. Skier4 different from skier3 preferred starting with gear 3 then switched between gear 2 and 3. He also preferred gear 2R compare to 2L but skier3 used gear 2L more than him. Skier4 was faster during this uphill, where we could ask if this was because he is faster during this kind of uphill which was long and then right after one very steep uphill.

Table 4.33: Fifth Section: 2610m

<table>
<thead>
<tr>
<th>Laps</th>
<th>Speed (m/s)</th>
<th>Speed (m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lap1</td>
<td>8.50</td>
<td>8.64</td>
</tr>
<tr>
<td>Lap2</td>
<td>8.56</td>
<td>9.06</td>
</tr>
<tr>
<td>Lap3</td>
<td>8.08</td>
<td>8.82</td>
</tr>
<tr>
<td>Lap4</td>
<td>9.78</td>
<td>9.49</td>
</tr>
</tbody>
</table>

*In overall skier4 was faster on this section*
Chapter 5

Evaluation
5.1 Performance of the system

Overall, this project has been a resounding success. We were able to design, build and test a system that is capable of communicating with multiple wireless sensors, stream the data collected to the cloud, whereupon it can be viewed either in real-time or after the fact. Once the data is stored on the server, it can be used to generate a wide variety of reports that help skiers and coaches understand the relationship between various Key Performance Indicators (KPIs).

Using a smartphone attached to a skier or a sensor belt, Skitracker is able to collect high-frequency accelerometer data which is then provided to the technique analysis tool developed at SICS. This enables experiments and classification with this tool to be conducted faster and easier than was possible before. Additionally, the results of this analysis can be seen in our web interface, which allows users to quickly view information about the technique with a minimum of effort.

Testing of the system with Ski Team Sweden was also very positive, with one of the main focuses being the stability of the system. During more than 50 total hours of testing, the Android application did not crash once, even when presented with issues such as poor GPS signal and sensors being disconnected. Regarding GPS signal, it is recommended that the smartphone be mounted in such a way so that it has a clear view of at least half the skyline to avoid losing signal. Although the system keeps recording other data when no GPS data is available, it does make interpreting the results harder, as the data lacks context in terms of position and speed.

The skiers reported that the equipment fitted to them was no more obstructive to their movements than the normal pulse belts they typically wear, which indicates that such a system would be feasible to implement on a larger scale. One Ski Team Sweden coach commented that it was the first time that they had seen real, concrete numbers regarding skiing technique.

One cause for disappointment was that, after a week of collecting data, the prototype pole force sensor stopped working correctly, likely due the strain gauge amplifier being faulty. Before this happened, however, we did managed to collect some test data, which shows that the concept works. We believe that going forward, this is a piece of equipment that can be developed into an extremely important analysis tool for poling technique.
5.2 Future work

Being a long-term project, development of Skitracker will continue for some time to come. Whilst this project has produced a stable, extensible framework that is capable of gathering almost any kind of sensor data and sending it to the cloud, there is still much work to do.

In the short term, we aim to incorporate classification of classical technique into Skitracker, which will mean that the system will be able to classify and analyse all major techniques. Looking further ahead, it is possible that the technology behind Skitracker could be used to produce a simple smartphone application that amateur skiers can use to improve their technique. The learning experience could perhaps be augmented by allowing the skiers to compare their performance to an elite-level skier. A social aspect could also be introduced, with the user being encouraged to compete against their friends.

Another goal could be to try and improve the experience for spectators at cross-country skiing events. Due to the nature of the sport, spectators are often able to see less of the skiers when standing by the track than someone watching on television. If each skier had the same set of ultra-light sensors mounted on the bodies and equipment, this data could be relayed in real-time directly to a spectator’s smartphone. Going a step further, the system could conceivably gather data, such as video clips or temperature data, from spectators’ devices, and stream this data to the cloud or other users.

Away from skiing, this technology has many applications. The collection of sensor data from sensors placed on the athlete or their equipment applies to almost every sport. For example, our communication API has already been used in the development of a system that records sensor data from a downhill mountain bike, and we are currently investigating the possibility of using our system to record data from runners, ice-hockey players and even racehorses.

As wearable sensors become smaller, cheaper and more energy-efficient, we will see a dramatic increase in the amount of sensor data being recorded by amateur and professional users alike. Skitracker is just one example of how the ‘Internet of Things’ is changing modern sport for the better.
5.3 Division of work

Razee Hussein-Jamal

- Web interface
  Setting up the tool and handling some basic functionalities (including the user account system), features and design.

- Android
  User interface and experience, communication with Bluetooth sensors.

- Shimmer platform
  Creating periodic tasks, handling Bluetooth communication and reading from the ADCs.

- Testing
  Data analysis, including detailed analysis of data collected in Toblach.

Tom Homewood

- Web interface
  User interface and performance optimisation, integration of Track API.

- Android
  Processing and storage of incoming sensor data, integration with Track API, multi-threading.

- Track API
  Design, implementation and testing of the API.

- External sensors
  Designed and built the prototype pole force sensor.
Bibliography


[16] Syunji Yazaki and Toshio Matsunaga. Elevation of activity level of daily life based heart rate and acceleration.
6.1 The Track API: A user’s guide

6.1.1 Introduction

The track API is a HTTP-based system that is used to interact with sensor data from remote devices. It can handle uploading, downloading and real-time streaming of sensor data, and can be used to present this information in a variety of formats, either in a web browser, or natively on a remote device. Data are grouped into time series called tracks, which can represent sets of sensor data over a time period.

The API consists of three major parts, which are used in conjunction with one another. These are:

1. The accounts layer.
   This is a MySQL database which contains details of all users registered with the system, and a set of php functions. Users can create accounts and login / logout using their user ID and password. Each user has an API key associated with their account, which is used for all API access, and provides a security mechanism, allowing users to control who can view their recorded data.

2. The data layer.
   This is the intermediate level of the system, and can be used to get raw data from a track, upload data to a track and to edit track the track’s properties. The data output can be customised in a number of ways, which will be detailed later. This layer can be accessed from any platform that supports HTTP requests.

3. The visual layer.
   This layer is used to take data from the data layer and present it in graphical format in a web browser. This can mean plotting a track on a map, or graphing any number of variables against time or against each other. This layer can be used only in a web browser.

The API is designed in such a way that the client does not need to interact with the data layer directly. Any call to the visual layer will use the data layer automatically. However, if the client wishes to retrieve raw track data (for use in a native application, for example), then the data layer can be used on its own.

This document provides a concise description of the API’s capabilities, together with a step-by-step guide to installing and using the API in your system. The example commands listed here are intended for use with the files provided.
6.1.2 Setting up the track API

The track API needs to have the following software components installed on the server where it will be used in order to function correctly:

- Apache webserver (version 2 or later)
- PHP (version 5.4 or later)
- MySQL (version 5.0 or later)
- phpMyAdmin (version 3 or later)

Once these components are up and running, follow the instructions below to set up the track API:

1. Extract the supplied .zip file somewhere. Inside are the files for required by the track api, plus an empty database which we will use to set up the user account system.

2. In phpMyAdmin, create a database that you wish to use for your system (you can use an existing database if you have already created one).

3. Select this database and choose the import option. Choose the file database.sql from the extracted folder and import it. This will import two new tables into your database, called user_profile and user_tracks, which will contain details of users and tracks respectively. A test user account and a sample track are included in the two tables for reference. There are example commands listed later in this document that use this test account and track.

4. Copy the folders accounts, common, track and users to a folder on your web server. This can be any folder you choose, but it is recommended the files are located somewhere within your website’s root folder. Note that the files in the accounts folder are not a complete website, but simply contain the required function calls for user registration, login and logout. You will have to integrate these functions into your own website. These three folders must be kept together for the system to work.

5. In the common folder, open mysql.php and change the four variables at the top of the file to match the details of the mysql database you created.

6. Open the file trackConfig.php (located in the ‘track’ folder), and add the data fields you want to use in your system to the array $validTypes. Each entry in this array is a key-value pair, where the key is a three-character code that the system will use to identify the data field, and the value is the human-friendly name for the data field, which is displayed on graphs and other UI elements. Add entries for these data fields in the array $dataFieldUnits, using the same three-letter code as the array index. Each entry in this array defines the units that are used for this data field, in the three main units systems (SI, metric and imperial - in that order). Finally, if the data field is one where the units change between the three unit systems (for example speed), then enter a pair of values in the array $unitConversionCoefficients, again using the three-letter code as the index. This pair of values represents the conversion coefficients required to convert the field from SI units to the metric and imperial systems respectively. Since data is always stored in the SI system, a coefficient for this system is not required. In the code provided, five data fields have already been
defined. The first three, latitude (lat), longitude (lon) and speed (spd), are examples, and can be removed if you wish. Two further fields, time (tim), and all fields (all), are defined, and these should not be removed. Time is included because ‘time’ will almost always be used as a field, and ‘all’ is a special field that can be used to refer to all defined data fields. These two items should not be removed from this array, and should always be the last two items in the array.

6.1.3 Using the accounts layer

This layer handles user account creation and authentication, and must be implemented in your system in order for the API to function correctly.

What you need to implement in your own system is a script that registers a user (taking its data from an HTML form) in the user_profile database table. For the API to work, an api key must be defined for each user. This is a string of up to 50 characters in length. See section 3.1 for more information about the api key. A good generator for this key in php is the function sha1(input).

Additionally, each user must their own folder in the users folder. The name of this folder must match their UserId field in the user_profile table. Within this folder there must be another folder, named tracks. This is where the user’s tracks will be stored, so it is required in order for the API to function correctly.

Aside from these two requirements, you are free to implement your account system as you wish.

Security and track visibility

Every track in the system has a field in the database called visibility, which determines who can view the track. There are three possible values for this field, and these are:

- **private**
  The track may only be viewed by the user whose userId matches the track’s ownerId field.

- **users**
  The track may be viewed by anyone with an account in the system.

- **public**
  The track may be viewed by anyone.

Access control is managed by allocating a unique api key to each user account. This key is required when accessing data from tracks with visibilities users or private. Users should not share their api keys at any time.

6.1.4 Using the data layer

Interactions with the data layer are performed using HTTP GET requests. Some commands can be used without authentication, but only public data will be returned in this case. Commands that do require authentication need to have a valid userId – apiKey pair passed to them in order to function correctly. Tracks are stored on the webserver in JSON format in the user’s own folder, and are also represented as entries in the user_tracks
table. If an error or warning occurs when a command is run, a JSON object is output, in the form:

{"error": "error_message"} or {"warning": "warning_message"}

You must therefore check for these when processing returned data. Some commands return an ACK upon successful execution, in the form:

{ "ack": "command_completed_successfully"}

The diagram below illustrates the lifecycle of a track in the system (blue actions are user interactions):

During its lifetime, a track is created, updated zero or more times, then archived. Data may be retrieved from the track when it is either running (i.e. realtime) or when it is archived (a past track).

This section will cover all available commands found in the data layer, and examples of how to use each command will be provided.
Creating a track

To start a track, perform an HTTP GET request to the following location:

http://<your_website_address>/track/connect.php?userId=<string>&apiKey=<string>

Required parameters:

userId the id of the user connecting
apiKey the api key for the user
sharingMode the visibility of the track. Permitted values are 'private', 'users' and 'public'
providedTypes a comma-separated list of the data type codes that this track contains (e.g. 'lat,lon,spd')

Optional parameters:

name the name to give the track
deviceName a string describing the device being used for the track, for example ('Asus Nexus 7')

Return values:

ACK containing the id of the newly created track on success
user_details_not_specified if either userId or apiKey is not provided
user_details_incorrect if userId or apiKey was invalid
sharing_mode_not_specified if sharingMode was not provided
invalid_sharing_mode_specified if sharingMode is not one of the permitted values
provided_types_not_specified if providedTypes was not provided
data_fields_invalid if one or more of the types in providedTypes is not in the array $validTypes
user_already_connected if the user already has a track running on the server
could_not_create_file if the track file could not be created
error_creating_track_in_database if the track’s details could not be saved in the database

Example using the test account provided:

http://<your_website_address>/track/connect.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bd29c36911bc030

76
Adding data to a track

To update a track's with new data, perform an HTTP GET or POST request to the following location:

http://<your_website_address>/track/update.php?userId=<string>&apiKey=<string>&newData=<string>

Note that in the above example the newData parameter is provided as a GET variable. *If the value for newData is long, then it may also be passed as a POST variable to avoid the URL becoming too long. In this case the GET variable newData should be omitted from the URL.*

**Required parameters:**

- **userId** the id of the user connecting
- **apiKey** the api key for the user
- **newData** the new data for the track, in JSON array form (see Data Format section below)
- **trackId** the id of the track

**Return values:**

- **ACK** on success
- **user_details_not_specified** if either **userId** or **apiKey** is not provided
- **user_details_incorrect** if **userId** or **apiKey** was invalid
- **track_id_not_specified** if the **trackId** was not provided
- **track_with_id_track_id_not_found** if the provided **trackId** was not found in the system
- **update_data_not_provided** if no new data was provided
- **data_invalid** if the data provided is not valid JSON
- **data_fields_invalid** if the new data fields do not match those defined in the config file

**Example using the test account provided:**

http://<your_website_address>/track/update.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&newData=[{"lat":"55.1284","lon":"17.4376"}]

**Data format:**

Data must be in JSON array format, with each point a JSON object. If just one point is provided, it must still be in a JSON array. Below is an example JSON array containing two points, with the fields ‘lat’ and ‘lon’ provided:

[{"lat":"55.7358", "lon":"17.5629"},{"lat":"56.3971", "lon":"17.8236"}]

Where applicable, data must be provided in SI units (m,s,kg etc. etc.). Time is provided in milliseconds since 00:00 on Jan 1st 1970 (Unix time).
Archiving a track

When a track is complete, it needs to be archived (stopped), after which it will no longer be updateable using the update command. To archive the user’s current track, perform an HTTP GET request to the following location:

http://<your_website_address>/track/disconnect.php?userId=<string>&apiKey=<string>

Required parameters:

- userId: the id of the user connecting
- apiKey: the api key for the user
- trackId: the id of the track

Return values:

- ACK: on success
- user_details_not_specified: if either userId or apiKey is not provided
- user_details Incorrect: if userId or apiKey was invalid
- track_id_not_specified: if the trackId was not provided
- track_with_id_'track_id' _not_found: if the provided trackId was not found in the system
- user_alreadyDisconnected: if user already did not have a track running on the server

Example using the test account provided:

http://<your_website_address>/track/disconnect.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030
Adding a log file to a track

Once a track is complete, log files contain high-frequency sensor data may be added to it using the `addLogFile` command. To start the process of adding a log file, perform an HTTP request to the following location:

http://<your_website_address>/track/addLogFile.php?userId=<string>&apiKey=<string>&trackId=<int>&fileName=<string>

Once a connection has been opened, the data in the log file needs to be uploaded using a stream. The compression scheme Deflate is supported for this stream upload, but not required in case such a compression scheme is unavailable on the device.

**Required parameters:**

- `userId`: the id of the user connecting
- `apiKey`: the api key for the user
- `trackId`: the id of the track
- `fileName`: the name of the log file, including its extension

**Return values:**

- **ACK**: on success
- **user_details_not_specified**: if either `userId` or `apiKey` is not provided
- **user_details_incorrect**: if `userId` or `apiKey` was invalid
- **track_id_not_specified**: if the `trackId` was not provided
- **track_with_id_track_id_not_found**: if the provided `trackId` was not found in the system
- **file_name_not_specified**: if no file name was provided
- **data_not_provided**: if no data was written to the connection
- **data_invalid**: if the data uploaded was not a valid json array
- **error_decompressing_data**: if the request header specified that the data is compressed but the data could not be decompressed
- **could_not_save_data**: if the uploaded data could not be saved on the server

**Example using the test account provided:**

http://<your_website_address>/track/addLogFile.php?userId=test@email.com&apiKey=d4c3d66fd0c36547a3c7a4c6bd29c36911bc030&trackId=1&fileName=phone_accelerometer.csv
Uploading a track file

With the Track API, it is also possible to upload an entire track file with one single command. This is useful if the track you wish to upload was recorded without a network connection for example, and needs to be synced to the server. To start the process of uploading a track file, perform an HTTP request to the following location:

http://<your_website_address>/track/addTrackFile.php?userId=<string>
&apiKey=<string>&sharingMode=<string>&typesProvided=<string>

Once a connection has been opened, the data in the track file needs to be uploaded using a stream. The compression scheme Deflate is supported for this stream upload, but not required in case such a compression scheme is unavailable on the device. Once a track file is uploaded, a backup is created immediately, so that any editing of the track file can be undone if desired.

Required parameters:

userId
the id of the user connecting
apiKey
the api key for the user
sharingMode
the visibility of the track. Must be one of public, users or private
providedTypes
a comma-separated list of the data type codes that this track contains (e.g. ‘lat,lon,spd’)

Optional parameters:

name
the name to give the track
deviceName
a string describing the device being used for the track, for example (‘Asus Nexus 7’)

Return values:

ACK containing the id of the newly created track on success
user_details_not_specified if either userId or apiKey is not provided
user_details_incorrect if userId or apiKey was invalid
sharing_mode_not_specified if sharingMode was not specified
invalid_sharing_mode_specified if sharingMode is not one of the permitted values
provided_types_not_specified if providedTypes was not provided
data_fields_invalid if one or more of the types in providedTypes is not in the array $validTypes
track_already_exists if a track with the same name already exists for the user
data_not_provided if no data was written to the connection
error_decompressing_data if the request header specified that the data is compressed but the data could not be decompressed
could_not_save_data if the uploaded data could not be saved on the server
error_creating_track_in_database if the track’s details could not be saved in the database
Example using the test account provided:

http://<your_website_address>/track/addTrackFile.php?userId=test@email.com
&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&trackId=1&fileName=
logFileName.csv&sharingMode=users&typesProvided=lat,lon,spd,ele,acc,brg
Updating a track’s lap points

A track may contain a set of so-called lap points, which are used to split the track into sections to allow easier and more powerful data analysis. Lap points are specified as the indexes of the points within the track file, so the tenth point in the track is expressed as point 9 (numbering starts at 0). These lap points may be updated using the Track API using the updateLapPoints command.

To set a track’s lap points, perform an HTTP GET request to the following location:

updateLapPoints.php?userId=<string>&apiKey=<string>&trackId=<int>&lapPoints=<string>

Required parameters:

userId the id of the user connecting. To set lap points, the user must be the track’s owner
apiKey the api key for the user
trackId the id of the track
lapPoints a comma-separated list of the indexes of the track points (eg ‘1,100,150’). If you wish to clear any lap points from the track, this may be set to an empty string

Return values:

ACK on success
user_details_not_specified if either userId or apiKey is not provided
user_details_incorrect if userId or apiKey was invalid
track_id_not_specified if the trackId was not provided
track_with_id ‘track_id’ not_found if the provided trackId was not found in the system
user_not_owner if the userid provided is not the owner of the track
lap_points_not_specified if lapPoints was not specified
error_updating_lap_points if the track file could not be updated

Example using the test account provided:

http://<your_website_address>/track/updateLapPoints.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&trackId=1&lapPoints=10,20,30,100
Automatic lap point detection

A track containing latitude and longitude data may consist of several laps around a track. In this case, the Track API is capable of detecting where new laps begin and adding lap points as necessary. This detection is automatically run when a track is archived if the track does not already contain any lap points. However, automatic lap detection may also be run at any time afterwards, and is triggered by performing an HTTP GET request to the following location:

http://<your_website_address>/track/runLapDetection.php?userId=<string>
&apiKey=<string>&trackId=<int>

Required parameters:

userId the id of the user connecting. To run lap detection, the user must be the track’s owner
apiKey the api key for the user
trackId the id of the track

Return values:

ACK on success
user_details_not_specified if either userId or apiKey is not provided
user_details_incorrect if userId or apiKey was invalid
track_id_not_specified if the trackId was not provided
track_with_id_track_id_not_found if the provided trackId was not found in the system
user_not_owner if the userid provided is not the owner of the track
error_updating_lap_points if the track file could not be updated
no_lap_points_found if no lap points were found

Example using the test account provided:

http://<your_website_address>/track/runLapDetection.php?userId=
test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030
&trackId=1
Updating a track’s name

A track’s name may be updated at any time, and is done by performing an HTTP GET request to the following location:

http://<your_website_address>/track/updateTrackName.php?userId=<string>
&apiKey=<string>&trackId=<int>&newName=<string>

Required parameters:

userId the id of the user connecting. To update the track’s name, the user must be the track’s owner
apiKey the api key for the user
trackId the id of the track
newName the new track name

Return values:

ACK on success
user_details_not_specified if either userId or apiKey is not provided
user_details_incorrect if userId or apiKey was invalid
track_id_not_specified if the trackId was not provided
track_with_id_'track_id'_not_found if the provided trackId was not found in the system
user_not_owner if the userid provided is not the owner of the track
track_name_not_specified if newName was empty
error_updating_track_name if the track’s name not be updated

Example using the test account provided:

http://<your_website_address>/track/updateTrackName.php?userId=
test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030
&trackId=1&newName=exampleName
**Restoring a track**

When a track is archived, a backup of the main track file is made automatically. If changes have made to the main track file that you wish to undo, you may restore the main track file from this backup. This is done by performing an HTTP GET request to the following location:

http://<your_website_address>/track/restoreTrack.php?userId=<string>&apiKey=<string>&trackId=<track_id>

**Required parameters:**

- **userId**: the id of the user connecting. To restore a track, the user must be the track’s owner
- **apiKey**: the api key for the user
- **trackId**: the id of the track

**Return values:**

- **ACK**: on success
- **user_details_not_specified**: if either userId or apiKey is not provided
- **user_details_incorrect**: if userId or apiKey was invalid
- **track_id_not_specified**: if the trackId was not provided
- **track_with_id,’track_id’ not_found**: if the provided trackId was not found in the system
- **user_not_owner**: if the userId provided is not the owner of the track
- **track_currently_active**: if the track is realtime and still being updated
- **could_not_restore_track**: if the backup file could not be copied to the track file

**Example using the test account provided:**

http://<your_website_address>/track/restoreTrack.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&trackId=1
**Requesting data from a track**

Data from a track may be requested using the script `trackData.php`. Any one or more of the track fields may be requested at the same time, and options such as the data format, start point and unit system may be specified. Additionally, the data fields may optionally be filtered using a set of functions and filters. Use the arrays `$validFilters`, `$fieldFilterWhitelist` and `$fieldRoundingWhitelist` in `filters.php` to control how fields are filtered.

To request data from a track, perform an HTTP GET request to the following location:

```
http://<your_website_address>/track/trackData.php?trackId=<number>&types=<string>
```

**Required parameters:**

- **trackId**: the id of the track you wish to access
- **types**: comma-separated list of 3 letter data field codes representing the data fields you wish to retrieve (e.g. ‘lat,lon,spd’)


Optional parameters:

userId
user Id of the user accessing the track (required when the visibility of the track is not public)

apiKey
api key for the above userId

format
the format in which to output data. Can be one of json, csv, or xml. These values are defined in the config file. If not set, defaults to json

download
boolean. If true, the data is output as a file for the user to download, if false the data is simply output as a plain text stream. Defaults to false if not set

start
index of the point to start from. Useful if you only want new data

limit
the maximum number of points to return. If not specified or set to 0, there is no limit

units
which unit system to use for the data (si, metric or imperial). Default is si

filters
comma-separated list of filters to apply to the data fields. Each filter consists of a 3-letter filter code, followed by a pipe symbol (\(\mid\)), and then a parameter for that filter. For example, if the filter avg is selected, then the parameter after the pipe symbol represents the sample size of the average. Supported filters are Average (avg), Weighted average (wav), Kalman (kal), First derivative (1df) and Second derivative (2df). If only one filter is specified, it is applied to all data fields. The filter codes must match those defined in the array $validFilters in filters.php

showLaps
boolean. If true, lap fields are output in points that are lap points. Defaults to false if not set

interpolate
the maximum number of points to output. This can be used to show fewer data points than the track contains. For example, setting interpolate to 500 for a track containing 1000 points will output every other point
Return values:

JSON, CSV or XML formatted data on success

- `track_id_not_specified` if `trackId` was not specified
- `track_not_found` if track with the provided `trackId` was not found in the database
- `user_details_not_specified` if track requires authentication and either `userId` or `apiKey` is not provided
- `user_details_incorrect` if `userId` or `apiKey` was invalid
- `user_not_owner` if the track's visibility is private and the `userId` provided does not match the user id of the owner of the track
- `no_data_fields_specified` if not data fields were specified in the `types` parameter
- `data_fields_invalid` if the data fields specified in the `types` parameter do not match the ones defined in `tracktrackConfig.php`
- `invalid_unit_system_specified` if the unit system specified is not one of `si`, `metric` or `imperial`
- `error_accessing_file` if the file could not be found at the path defined in the database entry for this track
- `error_validating_file` if the contents of the track data file are not a valid JSON array
- `start_timestamp_too_early` if the `start` timestamp specified is before the first point
- `start_timestamp_too_late` if the `start` timestamp specified is after the last point
- `end_timestamp_too_early` if the `end` timestamp specified is before the first point
- `end_timestamp_too_late` if the `end` timestamp specified is after the last point
- `start_timestamp_is_after_end_timestamp` if the `start` timestamp is later than the `end` timestamp
- `output_format_invalid` if the parameter `format` is not one of `json`, `csv` or `xml`
- `download_parameter_invalid` if the parameter `download` is not `true` or `false`
- `filters_invalid` if the parameter `filters` is not formatted correctly
- `filters_not_defined` if the one or more of the filters specified is not defined in the array `$validFilters` in `filters.php`
- `number_of_filters_does_not_match_number_of_fields` if the number of filters specified is not 1 or equal to the number of data fields requested
- `showLaps_parameter_invalid` if `showLaps` was something other than `true` or `false`
- `interpolate_value_invalid` if `interpolate` is less than zero

Example using the test account provided:

http://<your_website_address>/track/trackData.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&trackId=1&types=lat,lon,spd
Retrieving a track’s ID

Some applications may need to check if a track with a certain name exists on the server, and if so, what ID it has. This is vital for any application that is capable of recording tracks offline and wishes to sync these tracks to the server at a later date. You can retrieve a track’s ID by performing an HTTP GET request to the following location:

http://<your_website_address>/track/getTrackId.php?trackName=<string>

**Required parameters:**

- **trackId** | the id of the track

**Optional parameters:**

- **userId** | the id of the user connecting
- **apiKey** | the api key for the user

**Return values:**

A JSON object containing the track’s id and the *userId* of it’s owner

- **track_name_not_specified** if *newName* was empty
- **track_not_found** if a track with the specified name could not be found, or the track was found but the provided *userId* and *apiKey* do not allow access to it. The same error is output in both cases for security

**Example using the test account provided:**

http://<your_website_address>/track/getTrackId.php?trackName=trackName
Retrieving a track’s data fields

From time to time, it is useful to be able to retrieve a list of the data fields that a track contains. The command *trackFields* can be used to get information about each data field in the track, including its full name and units. This is done by performing an HTTP GET request to the following location:

`http://<your_website_address>/track/trackFields.php?userId=<string>&apiKey=<string>&trackId=<track_id>`

**Required parameters:**

- **trackId** the id of the track

**Optional parameters:**

- **userId** the id of the user connecting
- **apiKey** the api key for the user
- **showTypeUnits** boolean. If true, the units for the data fields are output as well as the name

**Return values:**

A JSON array containing details of the data fields in this track. See below:

- **user_details_not_specified** if either *userId* or *apiKey* is not provided
- **user_details_incorrect** if *userId* or *apiKey* was invalid
- **track_id_not_specified** if the *trackId* was not provided
- **track_with_id_track_id_not_found** if the provided *trackId* was not found in the system
- **user_not_owner** if the userid provided is not the owner of the track
- **track_name_not_specified** if *newName* was empty
- **error_updating_track_name** if the track’s name not be updated

**Example using the test account provided:**

`http://<your_website_address>/track/trackFields.php?trackId=1&userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030`

This will output:

```json
[{
  "shortCode": "tim",
  "fullName": "Time"
},
{
  "shortCode": "lat",
  "fullName": "Latitude"
},
{
  "shortCode": "lon",
  "fullName": "Longitude"
},
{
  "shortCode": "spd",
  "fullName": "Speed"
},
{
  "shortCode": "ele",
  "fullName": "Elevation"
},
{
  "shortCode": "acc",
  "fullName": "GPS accuracy"
},
{
  "shortCode": "brg",
  "fullName": "Bearing"
}]
```
Downloading log files

Once a log file has been added to a track, it may be downloaded using the `downloadLogFile` command. This command outputs the raw log file as a download, in its original form. This is done by performing an HTTP GET request to the following location:

```
http://<your_website_address>/track/downloadLogFile.php?trackId=<int>
&fileName=<string>&userId=<string>&apiKey=<string>
```

**Required parameters:**

- **trackId**: the id of the track to which the log file belongs
- **fileName**: the full name of the log file (including extension)
- **userId**: the id of the user connecting
- **apiKey**: the api key for the user

**Return values:**

- A complete log file, presented as an attachment to the HTTP response on success
- `user_details_not_specified` if either `userId` or `apiKey` is not provided
- `user_details_incorrect` if `userId` or `apiKey` was invalid
- `track_id_not_specified` if the `trackId` was not provided
- `track_with_id_'track_id'_not_found` if the provided `trackId` was not found in the system
- `user_not_owner` if the userid provided is not the owner of the track
- `file_name_not_specified` if `fileName` was empty
- `log_file_not_found` if the log file could not be found

**Example using the test account provided:**

```
http://<your_website_address>/track/downloadLogFile.php?trackId=1
&fileName=phone_accelerometer.csv&userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030
```
Requesting summary data from one or more tracks

The data layer API also provides a simple command for retrieving basic information about several tracks. This is useful if you wish display a list of tracks in your application. This command is not part of the track lifecycle, as it is purely for retrieving tracks summary data.

This command features support for custom fields. This means that fields particular to your application may be defined and used. For example, in an application monitoring a runner, you may wish to define custom fields for distance, time and average speed. These fields are defined in the file tracksCustom.php, in the array $validCustomFields. In this same file you should add a case statement for each field where indicated, inside which you should place your code for generating the value of this field. There are three example custom fields defined to get you started.

This command is also able to sort the output but any of the fields (both standard and custom). To sort the results, provide the field code you wish to sort by in the parameter sort. The sorting order can be ascending or descending, and is set using the parameter sortOrder. If sort is not set, the results are sorted by creation time, with newest results first, and realtime tracks are placed above historical ones.

The parameters start and limit have can be used to specify a subset of results to return. So for example, if a result set contains 50 results, setting start to 10 and limit to 20 will return results 11 to 31 (The first result is number 0). These parameters are useful when displaying pages of results on a webpage or device.

This command can be used without a valid userId and apiKey, as tracks can be viewed by anyone when their visibility is set to public.
To retrieve track summary data, perform an HTTP GET request to the following location:

http://<your_website_address>/track/tracks.php

**Optional parameters:**

- **userId**
  - userId of the user (required to view tracks that are not public)

- **apiKey**
  - api key for the above userId

- **limit**
  - maximum number of tracks to output. If not set, all tracks are output

- **start**
  - the index of the first track you want to return. First value is 0. So to ignore the first 10 tracks in the result set, set this to 10

- **limit**
  - maximum number of tracks to output. If not set, all tracks are output

- **realtime**
  - if true, returns only tracks that are still ongoing (real-time). false also permitted, but is unnecessary as this is the default value

- **customFields**
  - a comma-separated list of custom fields to output, on top of the standard fields. These custom fields must be defined in trackCustom.php

- **sort**
  - a field code to sort the data by. For example, to sort by creation time, just specify ctd here. This field can be a standard or custom field

- **sortOrder**
  - which way to sort the data. Can be either asc (ascending) or desc (descending). Defaults to desc if not set

**Return values:**

- **JSON formatted data**
  - on success

- **no_tracks_found**
  - if no tracks were found that match the parameters provided

- **user_details_incorrect**
  - if userId or apiKey was invalid

- **invalid_limit**
  - if start is less than zero

- **start_value_out_of_bounds**
  - if start is greater than the number of tracks available

- **invalid_limit**
  - if limit is less than zero or greater than 100,000

- **invalid_realtime_value**
  - if realtime was set and not true or false

- **custom_fields_invalid**
  - if the fields specified in the customFields parameter do not match the ones defined in trackConfig.php

- **sort_field_invalid**
  - if sort does not match any of the core or custom fields

- **sort_order_invalid**
  - if sortOrder was provided and was not asc or desc

**Example using the test account provided:**

http://<your_website_address>/track/tracks.php?userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&limit=10
6.1.5 Using the visual layer

The visual layer is a set of HTTP GET commands that can be used to produce maps and graphs of data from tracks stored in the system using Google’s Maps and Charts APIs. These commands output complete HTML documents containing the elements requested. These documents can be incorporated into a website using the iframe HTML tag, where the src attribute is the HTTP GET command for the graph or map you wish to retrieve.

Visual layer elements may be static (suitable for archived tracks), or dynamic (suitable for running tracks). Dynamic elements are refreshed periodically by the browser. All visual layer elements require Javascript to be enabled in the client’s browser in order to function correctly.

There are three visual layer commands that can be run to display track data in different ways: map, graphLine and graphScatter. The latter two commands are implementation-independent, meaning that they will work for any variables defined in trackConfig.php. However, map can only be used when the variables lat and lon are defined, as maps require latitude and longitude data.

Visual layer elements retrieve their data from the data layer automatically, meaning that the client does not have to interact with the data layer directly; this is handled behind the scenes. This enables graphs and maps to be produced very easily, using just a single request.

All visual elements use a handler function call to a function in the parent document (i.e. the HTML document that contains the iframe. This handler is fired every time a data point in an element point is hovered over with the mouse cursor. This handler function calls the function newGraphIndex(pointIndex) in the parent document with the point index of the point that was hovered over. This means that if you define this function in your parent page, it will be called when the graph is interacted with, enabling you to run some code specific to your system. Additionally, when a graph or map point is clicked the function graphPointClicked(pointIndex) is called in the parent document to allow interaction. If these functions are not defined they will not be called, so the system will fail elegantly and not produce any fatal Javascript errors which would affect the subsequent operation of the page.

Some visual layer elements provide methods that allow interaction with the element itself, for example, adding or removing markers to a map. These are described below.
6.1.6 Displaying maps

Maps are displayed using the map command. This command shows a track on a map, with an optional extra variable graphed underneath the map. If this variable is shown, it is linked to the map so that when the mouse cursor is placed over a graph point, the corresponding track point on the map is highlighted, and vice versa.

Lap points are shown using markers, which display the lap number when the cursor is placed over them.

Functions are also provided for adding and removing map markers, and for toggling the visibility of lap markers.

To show this element, perform an HTTP GET request to the following location:

http://<your_website_address>/track/map.php?trackId=<number>

**Required parameters:**

- **trackId**
  
  the id of the track you wish to access

**Optional parameters:**

- **userId**
  
  userId of the user accessing the track (required when the track is not public)

- **apiKey**
  
  api key for the above userId

- **realtime**
  
  enables or disables periodic updating of the map. Possible values: true or false. Defaults to false

- **updateInterval**
  
  interval between data refreshes in ms. If not set, defaults to $liveRefreshRateDefault (which is set in track-Config.php)

- **extraVariable**
  
  an extra variable to graph underneath the map (from same track as the map data)

- **units**
  
  which unit system to use for the extra variable’s graph. Permitted values are: si, metric or imperial. Defaults to si if not set

- **mapWidth**
  
  width in pixels for the map

- **mapHeight**
  
  height in pixels for the map

- **graphWidth**
  
  width in pixels for the graph (if $matchMapAndGraph-Widths is set to true in trackConfig.php, then this value will be overridden by the value of mapWidth)

- **graphHeight**
  
  height in pixels for the graph

- **trackLineColour**
  
  colour of the track line on the map (HTML string without the # symbol)

- **graphLineColour**
  
  colour of the graph line (HTML string without the # symbol)

- **fillUnderGraph**
  
  whether or not to shade under the graph line. Permitted values are true or false. Defaults to false
Return values:

HTML document containing a map and optionally a graph underneath on success

no_track_specified if no trackID was provided

track_line_colour_invalid if the provided trackLineColour was not a valid HTML colour code

graph_line_colour_invalid if the provided graphLineColour was not a valid HTML colour code

graph_fill_value_invalid if the provided graphFillColor was not valid

invalid_extra_variable_specified if the provided extraVariable was not found in $validTypes (set in trackConfig.php)

invalid_unit_system_specified if the provided unitSystem was not one of the three permitted values

Example using the test account provided:

http://<your_website_address>/track/map.php?trackId=1&userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&extraVariable=lat&units=metric&mapWidth=700&mapHeight=400&graphHeight=200&trackLineColour=ff0000&graphLineColour=0000ff

This will generate:
6.1.7 Displaying line graphs

Line graphs are displayed using the `graphLine` command. This command shows one or more line graphs of variables plotted against time, aligned vertically with respect to time. Each graph can interact with the other graphs, meaning that when the mouse cursor is placed over the graph point, the corresponding points on the other graphs are highlighted. As with the `map` command, this command also supports the `newGraphIndex` and `graphPointClicked` functions.

To show this element, perform an HTTP GET request to the following location:

```
http://<your_website_address>/track/graphLine.php?trackId=<number>&types=<string>
```

**Required parameters:**

- **trackId**: the id of the track you wish to access
- **types**: a comma-separated list of 3 letter data field codes representing the data fields you wish to retrieve (e.g. ‘lat,lon,spd’)

**Optional parameters:**

- **userId**: userId of the user accessing the track (required when the track is not public)
- **apiKey**: api key for the above userId
- **realtime**: enables or disables periodic updating of the map. Possible values: `true` or `false`. Defaults to `false`
- **updateInterval**: interval between data refreshes in ms. If not set, defaults to `$liveRefreshRateDefault` (which is set in `track-Config.php`)
- **title**: custom title. Leave blank to hide the custom title
- **units**: which unit system to use for the extra variable’s graph. Permitted values are: `si`, `metric` or `imperial`. Defaults to `si` if not set
- **graphWidth**: width in pixels for the graph
- **graphHeight**: height in pixels for the graph
- **graphLineColours**: comma-separated list of colour(s) for each graph’s line (HTML string without the # symbol) (e.g. ‘ff4332,6633ff’). Specifying just one colour sets all graph lines to this colour
- **fillUnderGraph**: comma-separated list of booleans that describe whether or not to shade under each graph’s line (e.g. ‘true,false,false’). Specifying just one value sets all graphs to this value
Return values:

HTML document containing one or more graphs on success
no_track_specified if no trackID was provided
graph_types_not_specified if types was not provided
cannot_request_all_graphs if types was set to all (this is invalid)
data_fields_invalid if one or more of the requested data fields was not found in $validTypes (set in trackConfig.php)
graph_line_colours_invalid if the provided graphLineColours was not a list of valid HTML colour codes
number_of_colour_codes_invalid if the number of provided colour codes was not 1 or equal to the number of types requested
graph_fill_values_invalid if the provided graphFillValues was not a list of booleans
number_of_fill_values_invalid if the number of provided colour codes was not 1 or equal to the number of types requested
invalid_unit_system_specified if the provided unitSystem was not one of the three permitted values

Example using the test account provided:

http://<your_website_address>/track/graphLine.php?trackId=1&userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bcd29c36911bc030&types=lat,lon&units=metric&graphWidth=700&graphHeight=200&graphLineColour=0000ff

This will generate:
6.1.8 Displaying scatter graphs

Scatter graphs are displayed using the `graphScatter` command. This command shows a scatter plot of any two variables. As with the `map` command, this command also supports the `newGraphIndex` and `graphPointClicked` functions.

To show this element, perform an HTTP GET request to the following location:

http://<your_website_address>/track/graphLine.php?trackId=<number>&types=<string>

**Required parameters:**

- `trackId` the id of the track you wish to access
- `types` comma-separated pair of 3 letter data field codes representing the two data fields you wish to plot against each other (e.g. ‘lat,lon’). The first field is plotted on the y axis and the second on the x axis

**Optional parameters:**

- `userId` userId of the user accessing the track (required when the track is not public)
- `apiKey` api key for the above userId
- `realtime` enables or disables periodic updating of the map. Possible values: `true` or `false`. Defaults to `false`
- `updateInterval` interval between data refreshes in ms. If not set, defaults to `$liveRefreshRateDefault` (which is set in `trackConfig.php`)
- `title` custom title. Leave blank to hide the custom title
- `units` which unit system to use for the extra variable’s graph. Permitted values are: `si`, `metric` or `imperial`. Defaults to `si` if not set
- `graphWidth` width in pixels for the graph
- `graphHeight` height in pixels for the graph
- `graphPointsColour` colour of the graph’s scatter points (HTML string without the # symbol) (e.g. ff4323)

**Return values:**

- HTML document containing a scatter graph on success
- `no_track_specified` if no `trackId` was provided
- `graph_types_not_specified` if types was not provided
- `cannot_request_all_graphs` if types was set to `all` (this is invalid)
- `data_fields_invalid` if one or more of the requested data fields was not found in $validTypes (set in `trackConfig.php`)
- `graph_points_colour_invalid` if the provided `graphPointsColour` was not a valid HTML colour code
- `invalid_unit_system_specified` if the provided `unitSystem` was not one of the three permitted values
Example using the test account provided:

http://<your_website_address>/track/graphScatter.php?trackId=l&userId=test@email.com&apiKey=d4c3d66fd0c38547a3c7a4c6bdc29c36911bc030&types=lat,lon&units=metric&graphWidth=700&graphHeight=700&graphPointsColour=0000ff

This will generate:

Note, the two data fields shown in the above scatter graph are not present in the system provided to you in the zip file, but are shown here because a scatter plot of latitude against longitude is really just a pair plot graph that resembles a map track. These two different variables are therefore shown to give a better demonstration of the graphScatter element.
6.2 Shimmer: An architecture description

6.2.1 Introduction

The Shimmer nodes are programmed using NesC language an approach to embedded systems that supports TinyOS design. The application is composed of components wired together to form an executable application. It’s known as a static language, therefore the resources are known during compile time and NesC language supports these 3 architectures:

- Component-based architecture
- Tasks and event based concurrency
- Split phase operations

How these architectures process during run-time:

1. The application runs two threads; tasks and hardware event handlers. Tasks run to completion and are not interrupted while events run to completion but may be preempted by a task or another event but both can be preempted by asynchronous events.

2. In the application the interface provider implements the commands and the interface user implements the events

3. The application code is composed of two components; module and configuration. The module handles the program logic and implementation of the application while configuration is the top-level and it’s used to wire the module to other components that the application requires

4. Configuration is connecting interfaces used by components to interfaces provided by other components

5. The application is event triggered based
### 6.2.2 Interfaces used in the application

**Boot**

First event triggered when system has been booted successfully

- `booted()`: Components can assume the system has been booted, it calls `init()` and `BTStdControl.start()`

**Init**

Initialise states

- `init()`: Initialise hardware states such as BluetoothInit and AccelInit

**Leds**

Handle the Leds

- `ledyOn()`: Turns led y ON
- `ledyOff()`: Turns led y OFF

**Timer<TMilli>**

Milliseconds

- `startPeriodic(y)`: Start counting until it reaches y ms
- `stop()`: Cancels the running timer
- `fired()`: Signals that the timer has expired

**shimmerAnalogSetup**

Handles the ADC reading using DMA

- `addAccelInputs()`: Enable ADC channel 3,4,5
- `addAnExInput()`: Enable ADC channel 0,7
- `finishADCSetup(sbuf0)`: Set initial DMA buffer
- `getNumberOfChannels()`: Get number of ADC channels
- `triggerConversion()`: Start transfer ADC reading to DMA buffer
- `stopConversation()`: Stop transfer ADC reading to DMA buffer
Mma_Accel

setSensitivity(RANGE_4_0G):
set range of accelerometer

wake():
start/stop reading ADC channels for internal Accel board

StdControl

control for different functions, starting required softwares

start():
start this component and its subcomponents

stop():
stop this components and its subcomponents

Bluetooth

handle Bluetooth operation

async event void connection-Made(uint8_t status):
asynchronous event will be triggered when Bluetooth connection is established successfully

async event void command-ModeEnded():
asynchronous event will be triggered when the command mode is enabled

async event void connection-Closed(uint8_t reason):
asynchronous event will be triggered when Bluetooth connection is closed

async event void dataAvailable(uint8_t data):
asynchronous event will be triggered when commands are received via Bluetooth

event void writeDone():
event indicates the end of Bluetooth data transmission

disableRemoteConfig():
disables remote configuration

Msp430DmaChannel
to use explicit arguments such as DMA

stopTransfer():
stop DMA transfer

async event void transfer-Done(error_t success):
asynchronous event will be triggered when transfer via DMA is done

repeatTransfer(void *src_addr, void *dst_addr,uint16_t size):
repeat DMA transformation using previous settings but new pointers and transfer size
6.3 Android: Bluetooth communication

6.3.1 Introduction

The mobile device has internal sensors and can be connected to external sensors via Bluetooth, the application has the feature to enable Bluetooth, scan for Bluetooth devices, and connect to devices using the Bluetooth APIs provided by the Android OS. Communication with Bluetooth sensors uses the Serial Port Profile (SPP) service that runs as part of the Bluetooth stack on the device. To inform the OS that we wish to use SPP for communication, we use the unique 128-bit id ‘00001101-0000-1000-8000-00805F9B34FB’, which is only used for SPP.

Android manifest: In the manifest the application is declared to access the Bluetooth APIs by setting permission to BLUETOOTH and have the functionality of turning on the Bluetooth radio device discovery and connecting to devices by setting the permission to BLUETOOTH_ADMIN.

6.3.2 Connecting to Bluetooth sensors

Several classes, objects and methods have been implemented to provide a successful and easily interacted Bluetooth feature. There are two dialogs on this part of the menu:

1. Sensors: lists the connected and paired sensor devices
2. Add sensors: starts scanning and lists defined sensors

Each class of external sensors have a prefix name defined in the mobile application:

- DEVICE_PREFIX_BIOHARNESS = BH
- DEVICE_PREFIX_POLE_SENSOR_LEFT = PL
- DEVICE_PREFIX_POLE_SENSOR_RIGHT = PR

Bluetooth sensors are defined in the mobile application with unique ID:

- BIOHARNESS = 10
- POLE_SENSOR_LEFT = 20
- POLE_SENSOR_RIGHT = 21

To proceed a Bluetooth action a default local Bluetooth adapter must be declared which allows the mobile device start device discovery, query a list of paired devices with their information, create a Bluetooth socket and listen to request connection from Bluetooth devices.

```java
// declares default local Bluetooth adapter object
BluetoothAdapter bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
// Before starting any process the Bluetooth must been enabled
// If not requests the user to enable it to proceed scanning
if(bluetoothAdapter!= null && !bluetoothAdapter.isEnabled()){ 
    enableBluetooth = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE); 
    startActivityForResult(enableBluetooth, ENABLE_BLUETOOTH);
}
```

Listing 6.1: Declaring default local Bluetooth adapter and showing a system Activity to enable Bluetooth
Listing 6.2: Start device scanning process

Initiating a Bluetooth device, allows the application to request connection to the device and request information from the device such as (device name, MAC address, class and bonding state). The discovered Bluetooth device objects are list on a dialog box and clickable to pair with, once the device objects are clicked on an alert dialog will pop up with information about the device.

Listing 6.3: Requests Bluetooth device’s information and creates a BluetoothService object to open a socket

Creating a BluetoothService object starts the process of establishing a connection and opening a Bluetooth socket, therefore the application needs to understand the state of the connection as there are threads running so declaring boolean values indicated connection state

public static final int STATE_NONE = 0
private static final int STATE_LISTEN = 1
private static final int STATE_CONNECTING = 2
private static final int STATE_CONNECTED = 3
ConnectThread is a thread that runs while attempting to make an outgoing connection with a Bluetooth device, connection either succeeds or fails.

```java
private class ConnectThread extends Thread{
private final BluetoothSocket mmSocket;// Declaring BluetoothSocket object

//BluetoothDevice declaration goes here
public ConnectThread(){ // BluetoothDevice object declaration goes here
    // Get a BluetoothSocket for a connection with BluetoothDevice
    try{
        // Declaration of a static final UUID_SPP
        // UUID_SPP = 00001101-0000-1000-8000-00805F9B34FB" for SPP connection
        tmpSocket = device.createRfcommSocketToServiceRecord(UUID_SPP);
    } catch(){//Logs error}
}

public void run(){
    // Cancelling discovery goes here
    try{
        // This is a blocking call and will only return on a successful connection or an exception
        mmSocket.connect(); // Attempts to connect to a remote device
    } catch () { // Logs error
        try{
            mmSocket.close(); // Immediately close this socket
        } catch () { // Logs error
            // Calls connectionFailed() method that indicates connection attempt failed, notify UI Activity and exits thread
        }
    }
    synchronized (BluetoothService.this) { // Resets the thread
        mConnectThread = null;
    }
    // To begin managing Bluetooth connection this method is called
    connected(mmSocket, mmDevice);
}
```

Listing 6.4: Connect Thread
Communicating with Bluetooth sensors

ConnectedThread is a thread that runs during a connection with a remote device, handles all incoming transmission.

```java
private class ConnectedThread extends Thread{
    // Declaration of BluetoothSocket object goes here
    private final InputStream mmInStream; // To read input streams

    public ConnectedThread(BluetoothSocket socket){
        InputStream tmpIn = null; // Temp input stream
        // Get BluetoothSocket input streams
        try{
            tmpIn = socket.getInputStream(); // Saves input stream from open socket
        } catch () {// Logs error
            mmInStream = tmpIn;
        }

        public void run(){
            // The message and it’s bundle for holding data values
            Message message;
            Bundle dataBundle = new Bundle();
            long timestamp;

            // Used for building the messages from the stream
            int character;
            String receivedMessage;
            StringBuilder buffer = new StringBuilder(30);
            // Keep listening to the InputStream while connected
            while(true){
                try{
                    // Keep reading from string while the character is not newLine(decimal 10)
                    while (((character = mmInStream.read())!=10)){
                        if(); // if it’s a valid text character (ASCII 32 to 126)
                    }
                    // Saves current timestamp in milliseconds since January 1, 1970
                    timestamp = System.currentTimeMillis();
                    receivedMessage = buffer.toString(); // Get the string out
                    // And clear the buffer, ready for the next message
                    buffer.setLength(0);
                    // Create a new message with the message code provided to this service
                    message = mHandler.obtainMessage(device_code);
                    // Add the timestamp to the bundle
                    dataBundle.putLong("timestamp", timestamp);
                    // Add the received message to the bundle
                    dataBundle.putString("receivedMessage", receivedMessage);
                    // Add the bundle and send the message
                    message.setData(dataBundle);
                    mHandler.sendMessage(message);
                } catch () {// Logs error and exits
                    }
            }
        }
    }
}
```

Listing 6.5: ConnectedThread()