La Trobe University
Bundoora, Victoria, Australia
Faculty of Science, Technology & Engineering
Department of Computer Science and Computer Engineering

Flexion Angle Analysis using Android and Inertial Sensors

Author:	Sampson J. Oliver
Student No:	17145547
Course:	B. Comp. Sci. / B. Sci. (Mathematics)
Author:	Thomas William Wright
Student No:	17287164
Course:	B. Comp. Sci. / B. Sci. (Chemistry)
Submission Date:	November 2014
Supervisors:	Mr Sameer Cooshna Dr Kinh Nguyen Dr Prakash Veeraraghavan Ms Mary Witten

Abstract

This study concerns the feasibility of using inertial sensors such as gyroscopes and accelerometers in low-cost, commercially-available devices such as smartphones, for the purpose of monitoring patients with knee injury. As part of the study, the design and implementation of a prototype for an aid (involving hardware and software components) for patients with knee injuries is given. The device is required to record how much a patient bends their knee during day-to-day activities, alerting the patient if the magnitude of the angle (flexion/extension distance, or, the amount of bend in the knee) is too great, or has otherwise exceeded a set of predefined thresholds for various activities.

Throughout this study we discuss viable technologies, both hardware and software, for the implementation of the prototype system. The prototype is envisaged to leverage sensors on two smartphone devices; one attached above the knee and one below the knee. From the data gathered on each device, the angle of bend of the knee is calculated. As further proof-of-concept for use in fields of health science and physiotherapy, the system implements further software to allow predefined actions to be taken based on the determined angle, as well as allow for a range of analysis on the gathered data via a server implementation. The analysis should be able to be viewed on a device immediately after use, or else on a repository of such data over a period of time.

This document begins by discussing related research and implementation, before introducing the requisite technologies for addressing the targeted problem domain. We then discuss the chosen implementation of the system and present the results of the implementation.

Acknowledgements

We would like to thank Mr Sameer Cooshna for always being available to help or provide insight at a moment's notice, and Ms Mary Witten for her expertise and insight into preparing this document, and as ever for her words of encouragement.

Special thanks go out to the Mathematics Department of La Trobe University, including Dr Marcel Jackson, Dr Yury Nikolayevsky, and Mr Kevin Bicknell for their input and confirmation of ideas in working with orientation data.

Additional thanks go to Dr Eddie Custovic and Mr Philip Pawlowski, of the La Trobe Engineering and Physics departments respectively, for providing materials and equipment necessary to construct suitable environments for recording test data. Further thanks go to Dr Eddie Custovic for providing use of his laboratories and research workspaces.

License

This project, including all source code and resources, is issued under the MIT license. The source is available on GitHub (at https://github.com/sampsonjoliver/latrobe-datacapture-dir). Any derivative works referencing the research and implementation discussed in this document or implemented in the source code must do so by including this license and giving due credit to the original authors. The MIT License is as follows:

Glossary

API	Application Programming Interface
BSON	Binary JSON
CRUD	Create, Read, Update, Delete; a methodology for RESTful services
CSS	Cascading Style Sheets
GPS	Global Positioning Systems
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
LED	Light Emitting Diode
MAC Address	Media Access Control Address
MEMS	Microelectromechanical Systems
NTP	Network-Time Protocol
OS	Operating System
ODM	Object Document Mapper
REST & RESTful	Representation State Transfer, a design pattern for web applications
SQL	Structured Query Language
UIL, BLL, DAL	User Interface, Business Logic, and Data Access Layers, respectively
URL	Uniform Resource Locator
UTC	Universal Coordinated Time
Wi-Fi	Wireless network technologies, operating under the 802.11 standard

Contents

List of Figures

List of Tables

1 Introduction

The system requires the use of motion tracking technologies in order to track the positioning and orientation of a patient's leg above the knee versus below the knee. From this we attempt to determine the difference in position and orientation in order to calculate the angle of flexion/extension (that is, the magnitude of the angular distance) between each orientation. This calculation may then be applied to many fields, including motion capturing and tracking in entertainment industries, health science industries in physiotherapy, and so on.

The system is intended as a proof-of-concept that targets the health science field, in a medical context. The system is intended to use low-cost, commercially available hardware. The system is therefore to track the angle of flexion/extension across two sensor nodes, implemented using commercially available smartphone devices, that send the data to a web server for storage and analysis.

We hence present a brief discussion on pre-existing motion tracking technologies that may be applied to the target domain.

1.1 Motion Tracking Technologies

Numerous motion tracking technologies currently exist, each leveraging different physical principles in order to operate. The involved principles afford each technology specific advantages and limitations, which are outlined below for the various technologies.

Optical systems operate by measuring the light received by cameras from strategically placed light sources. This data can be computed to calculate the position and orientation of the target. The light sources can act by reflecting existing ambient light (e.g. reflectors or uniquely-coloured surfaces, known as passive sources) or by emitting generated light (e.g. light-emitting diodes [LEDs], known as active sources). In an optical system, the light sources may be placed upon the moving body, with the cameras in the surrounding environment, or the opposite. The relative properties (dimensions, weight, fragility, etc.) of the light sources compared to the cameras affects the suitability of each configuration for different applications.

Optical systems are able to capture highly accurate positional information at very high sample rates. The flagship T-series$^{\text{TM}}$ system from Vicon Inc. is capable of sampling at 120 Hz, with positional errors of only 1 mm [1]. However, optical systems are typically costly and involve a laboratory or other prepared environment, as they require components to be carefully positioned around the target. The other major drawback of optical systems is that there must be a clear line of sight between the light sources and the cameras. As such, optical systems require multiple cameras to capture the target from different angles, and the area of motion must be free of any obstructions. Both of these requirements can limit the usefulness of the systems under particular circumstances.

Mechanical systems typically consist of rigid exoskeletal limbs that are attached to the target. As the target manipulates the limbs, potentiometers in the joints measure the joint angle directly. This form of measurement differs from other modern techniques, in which joint angle is calculated from the position and orientation of known points on the target. As such, these mechanical systems provide very accurate measurements of joints and pose, but commonly lack direct positional information. It is important to consider that while the accuracy of measurements is very high, the wearing of a mechanical system is very likely to interfere with the natural movement of the target. For example, the Gypsy 7^{TM} system from MetaMotion is capable of measuring joints at 0.125 degree resolution, but weighs approximately 4 kg and monitors 14 joints of the body [2].

Magnetic systems utilise magnetometers to measure the magnetic field passing through the sensors, and leverage the magnetic field of the earth to coordinate and provide relative position and orientation data. Magnetic sensors can be manufactured at a small size, provide high sampling rates, and are not subject to occlusion like optical systems. However, inaccuracies in measurements can be introduced by the presence of electronic devices or ferrous materials in the surroundings, and by the natural variations in the magnetic field of the earth.

Acoustic systems operate by analysing time-of-flight data for brief ultrasonic pulses. Similar to optical systems, acoustic systems require the use and placement of both emitters and receivers. Due to the nature of audio waves in regards to navigating obstacles, acoustic systems are capable of tolerating occlusion to a greater extent than optical systems, but often at the sacrifice of accuracy. As echoes must be allowed to die out between pulses, the sampling rate of acoustic systems can vary from 10 Hz to 200 Hz. Furthermore, speed of sound can be affected by temperature and humidity amongst other environmental factors, which can introduce inaccuracies.

Inertial sensor technology refers primarily to the use of accelerometers and gyroscopes, but in modern applications can extend to include the use of magnetometers also. An accelerometer measures proper acceleration (popularly measured in terms of g-force) along a single axis. A gyroscope exploits the principles of angular momentum to measure change in rotation. Inertial sensor systems were first applied for use in inertial navigation of rockets and later aeroplanes, but at this time were unsuitable for small scale motion tracking applications due to the size of the combined sensor and associated computer system.

The development of MEMS (Microelectromechanical Systems) brought about the modern inertial measurement units that are effective at motion tracking of objects on the scale of human limbs due to vastly decreased size. A modern inertial measurement unit typically consists of three accelerometers and three gyroscopes, with each set organised orthogonally to allow for sensing in three dimensions. A popular example of a modern inertial measurement unit is the Opal^{TM}. Weighing less than 22 grams, with dimensions of 48.4x36.1x13.4 mm, Opal^{TM} units are light enough to be attached to the limbs without discomfort or restriction of natural movement. MEMS sensors are able to be sampled at very high rates, such as the Opal^{TM} unit which samples at 1280 Hz.

Inertial systems do not require line of sight between any components, making them more suited to environments involving obstructions. Unlike magnetic or acoustic systems, inertial sensors do not suffer interference from nearby devices. The primary drawback of inertial sensors is their tendency to develop increasing error in accuracy over time, known as drift error. Drift error is introduced in two ways: small inaccuracies in the measurements made by the sensors themselves, and the need to perform mathematic integration over the collected data.

2 Literature Review

Since developments in MEMS technology has allowed the size of accelerometers and gyroscopes to be greatly reduced, inertial sensors have become a subject of interest for motion tracking applications. This section reviews a selection of recent relevant studies, focusing on the technologies and techniques used.

Williamson et al. used both a combined accelerometer-gyroscope system and an accelerometer-only system to measure the joint angle and angular velocity of the knee, comparing against measurements taken using a goniometer. He determined that the combined system utilising gyroscopes complemented by accelerometers to correct drift was more accurate than the accelerometer-only system [3].

Roetenberg conducted a series of experiments to investigate inertial sensors and the benefits of combining them with existing technologies such as magnetic and optical tracking [4]. His results quantified sensor drift of gyroscopes, showing that integration of noisy gyroscope data resulted in a drift of $10 - 25$ degrees after 1 minute [4, p. 31]. His results also showed the significant interference incurred by magnetometers from nearby ferromagnetic objects [4, p. 44].

Vlasic et al. developed an acoustic-inertial hybrid system that utilised ultrasonic time-of-flight data as a complement to the inertial data in an attempt to correct drift error. Ultrasonic pulses emitted from the torso of the target were recorded by microphones built into the inertial measurement units, and the time delay calculated. The recorded data was combined using an Extended Kalman filter, and was shown to be successful at reducing the drift commonly observed in purely inertial systems. However, a significant limitation of the system was the inability to correct drift in global translation and rotation. Also it is important to note that due to physical limitations in the hardware implementation, high-impact motions such as jumping or kicking could not be analysed [5, p. 7-8].

Using a system of only two inertial sensors placed on the upper and lower arm, Zhou et al. were able to effectively estimate wrist, elbow and shoulder position for a set of slowly executed tasks such as mimicking drinking or flexing the elbow. The experiment did not utilise a Kalman filter or equivalent for correcting drift in the estimations, rather relying on a simpler process of analysing Euler angles from combined sensor data against a kinematic model of the arm to distinguish between noise and actual movement [6]. Results showed a high correlation with estimates from the reference optical tracking system [6, p. 10], however this was for highly controlled motions at slow speeds.

El-Gohary also explored a pure inertial approach to motion tracking of the human arm using two inertial sensors placed on the upper- and lower-arm. The system utilised the Unscented Kalman filter, which improves upon the Enhanced Kalman filter by removing the need to linearise the system equations [7, p. 91]. Approaches explored in this system to reduce the effect of sensor drift on joint angle estimates were the incorporated modelling of random sensor drift, zero-velocity updates, and restriction of joint rotations deemed prohibited by the system's kinematic model of the arm. Results showed that these techniques were all effective at increasing accuracy [7, p. 103-104].

3 Proposed Technologies

The proposed technology for this implementation is to utilise consumer-grade Android devices as inertial sensor units for the purpose of measuring orientation data. The devices will be equipped with three-axis accelerometers, gyroscopes, and magnetometers to allow for inertial sensing in three dimensions. A minimum of two devices is required in order to calculate the required knee joint angle. Thus Bluetooth will be used for inter-device communication. As the Android operating system is capable of handling sensor data sampling and controlling the required Bluetooth communication, the application logic is to be developed as an Android native application to leverage this provided functionality.

In this section, each of the above requirements is discussed in relation to the problem domain.

3.1 Inertial Sensors

The operating principles and characteristics of modern inertial sensors can be found in subsection 1.1.

The properties of modern MEMS inertial sensors such as compact size, affordable cost, high sample rate and, lack of occlusion or interference make them highly suited to human motion tracking. However, there exists one significant drawback that must be addressed in order to utilise them effectively–drift error. Inertial sensor systems calculate position and orientation by advancing a previously determined state by the measured change over time, a process known as "dead reckoning". As this process continues over many steps, seemingly minor inaccuracies in measurements accumulate and result in significant inaccuracy. This accumulation is illustrated by Figure 1.

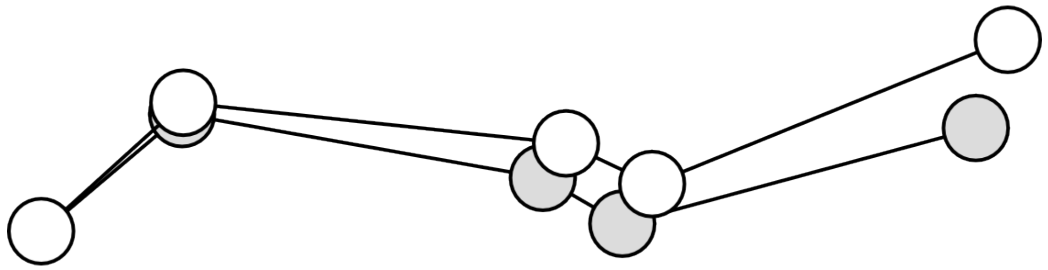


Figure 1: Dead Reckoning Inaccuracy - Accumulated error of dead reckoning process shown for measurements of displacement and orientation with a 5% positive bias. White path shows calculated position calculated by dead reckoning, grey path shows actual position at each sample point.

A number of algorithms exist (a commonly known example being the Kalman filter and associated variations) that can be applied to the amalgamation of sensor data streams to reduce noise and to produce a more statistically optimal estimate of the true sensor states [8]. To increase the effectiveness of these techniques, inertial measurement units may employ magnetometers to complement the accelerometric and gyroscopic data. Typically organised orthogonally in the same fashion as the other sensors, the trio of magnetometers measures orientation relative to a known magnetic field such as that of the earth. The proposed Android platform provides implementations of the above techniques to improve reliability of sensor data. These are covered in subsection 3.4.

As has been stated previously, accelerometers measure instantaneous acceleration and gyroscopes measure change in rotation, while the data generally desired is position and rotation (relative to some reference). Acceleration is change in velocity per unit time and velocity is change in position per unit time, therefore integrations over the measured data are necessary to perform the desired transformations. Given that the measured data consists of a finite number of instantaneous sample points rather than a continuous function, the integration calculation must be approximated by a method such as a Riemann Sum. Figure 2 shows two examples of such an approximation, and illustrates how the accuracy of a Riemann Sum approximation relates to how finely the region is divided.

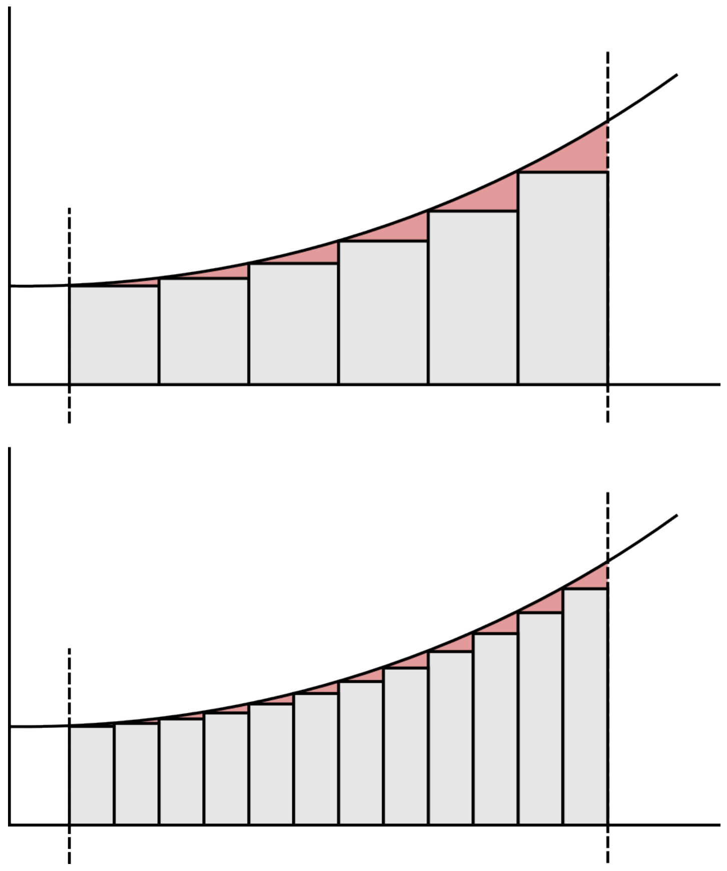


Figure 2: Riemann Sum for Approximation of Definite Integrals - showing two levels of region division with error indicated by shaded portions under the curve

As the region is divided into a greater number of intervals, the accuracy of the approximation increases. In terms of application to accelerometer and gyroscope measurements, the accuracy of position and rotation approximations increases proportionally to the sample rate, leading to reduced drift error overall.

3.2 Bluetooth

Bluetooth is a widely-used wireless communication standard that implements a packet-based protocol in a master-slave structure to form wireless personal area networks amongst participating devices. The technology is low-power and low-cost, and defines a uniform structure for a wide range of devices to connect and communicate with each other. Bluetooth uses roughly 3% of the power versus typical Wi-Fi solutions [9].

Common Bluetooth products implementing the Basic Rate or Enhanced Data Rate (BR/EDR) modes operate in the unlicensed industrial, scientific and medical (ISM) band at $2.4 - 2.485$ GHz, using a frequency-hopping, spread-spectrum, full-duplex signal [10]. Bluetooth adaptive frequency hopping (AFH) allows hopping among 79 frequencies at 1 MHz intervals, at a nominal rate of 1600 hops/sec to achieve more reliable communication in noisy environments, without receiving interference from common wireless (Wi-Fi) network signals. Common Bluetooth has a minimum mandated range of 10 m but may operate at up to 100 m, at a throughput of $1 - 3$ Mbit/s with a typical latency of 100 ms [11, 12, 13].

Devices implementing Bluetooth connect to each other via a process called pairing, which allows the devices to communicate securely through short-range, ad-hoc networks known as piconets; so-called due to the 3-bit address space of Bluetooth devices, which limits the maximum addressable size to 2^3 devices, allowing a maximum of one master and seven slaves. This functionality can be extended by allowing each client to participate in other piconets, forming a scatternet, which is a multiply-bridged piconet as shown in Figure 3.

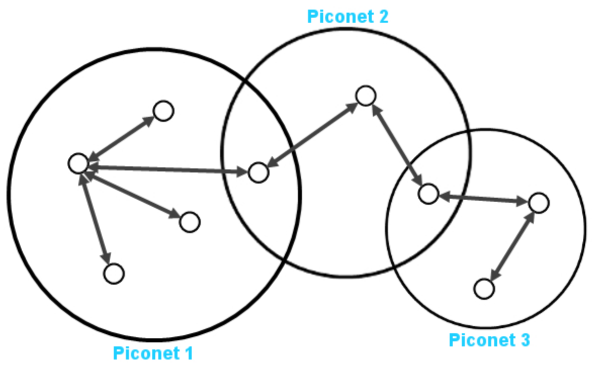


Figure 3: A Bluetooth Scatternet formed of Multiply-Bridged Piconets
Source: http://www.summitdata.com/blog/ble-overview/

Every Bluetooth device implements a native clock, referred to as the Bluetooth clock, derived from a free-running reference clock. The Bluetooth clock is intended to measure offsets between paired Bluetooth devices, where these offsets may be used to provide mutually synchronised temporary clocks. Devices in a piconet are synchronised to share the master's clock and frequency hopping pattern, where packet exchange between each device is based on this clock. It should be noted that this clock has no relation to time of day, and may be initialised to any value. The clock has a cycle of approximately one day, with a clock rate of 3.2 kHz [14]. The clock ticks at 312.5 μs intervals, where two ticks make up a slot of 625 μs, and two slots make up a slot pair of 1250 μs. Communication between a master and its slaves is done according to these clock ticks, with the master transmitting in even-slots and receiving in odd-slots. Packets sent between nodes may be 1, 3, or 5 slots long [14, 10]. Ringwald et al. present a method of providing high-accuracy synchronisation using the Bluetooth clock between devices in a scatternet, provided the implementing device has access to the Bluetooth clock.

It is worth noting that an additional Bluetooth specification introduced in the Bluetooth 4.0 Core Specification, announced in 2010, is Bluetooth Low Energy (BLE). Comprised of many non-backwards compatible changes, BLE utilises a 32-bit access address allowing for billions of connected devices in the traditional master-slave topology, over Common Bluetooth's 8-member piconets. BLE therefore provides a star-like network topology. It provides a typical latency of 3 ms, a possible range of 100 m, and 1 Mbit/s of data throughput [11]. Though BLE availability is presently more limited than Common Bluetooth, and is not supported in software by many operating systems (including Android prior to API level 18), its application to the project domain makes it a viable solution for future implementations.

Table 1 shows a comparison of Classic Bluetooth and BLE, with Wi-Fi included for comparison. Values were collected from whitepapers published by Cisco [15], Litepoint [16], and power draw data taken from Balani [9].

Standard	Classic Bluetooth	Bluetooth Low Energy	Wi-Fi (802.11n/ac)
Range (nominal-max)	10 – 100 m	10 – 50 m	100 m
Frequency Band	2.4 – 2.483.5 GHz	2.4 – 2.483.5 GHz	2.4 GHz; 5 GHz
Signal Rate	1 – 3 Mbit/s	1 Mbit/s	150 – 600 Mbit/s; 433 – 2600 Mbit/s
Throughput	0.7 – 2.1 Mbps	$<$ 0.3 Mbps	46 – 4900 Mbps
Channel Bandwidth	1 MHz	2 MHz	Mandatory: 20, 40, 80 MHz, Optional 160 and 80+80 MHz
Mechanism	Adaptive fast frequency hopping, FEC, fast ACK	Adaptive frequency hopping, Lazy Acknowledgement, 24-bit CRC, 32-bit Message Integrity Check	Dynamic frequency selection, transmit power control
Topology	Scatternet	Star-bus	ESS
Latency (non-connected)	100 ms	6 ms	N/A
Expected packet latency	100 ms	3 ms	$<$ 30 – 50 ms
Security	56 – 128 bit shared secret key, 16 bit CRC	128 bit AES with Counter Mode CBC-MAC	WPA2 256 bit encryption key
No. Channels	79	40	
Power Draw	2 – 24 mW		4 – 324 mW

Table 1: Comparison of Classic Bluetooth, BLE, and Wi-Fi

3.3 Network Time Synchronisation Protocol

In constructing logically coherent time-series datasets by amalgamating inputs from multiple disjoint sensor nodes, time synchronisation is required across each node within the network. While many devices maintain their own real-time clock, many of these devices do not measure ticks precisely to the second, and so become increasingly inaccurate over time. Further, it is not sufficient to rely upon each node's independently maintained clock, as inconsistencies between the devices even in the order of seconds is enough to cause significant precision error. It is therefore required that each node be time-synchronised such that each sensor can provide data time-stamped to a global time scale. The accuracy of synchronisation must be such that the time latency between slave and master nodes does not yield significant error in the resultant time-series dataset.

Internet time service providers and GPS time both provide possible external synchronisation methods, allowing for each sensor to sync against the same source. These methods each suffer inherently from the method in which the time must be retrieved, however, as the device clock must synchronise against a target clock that may take many seconds to reach, hence yielding potentially stale data. The greater and more variable the latency, the greater the order of clock error. That is, if two sensors s_1 and s_2 simultaneously request the current time from a provider p at time t_0, then if s_1 receives a response after 100 ms (define this as $t_1 = t_0 + 100$) and hence synchronises its clock to t_0, while s_2 then receives a response after 200 ms (define this as $t_2 = t_0 + 200$) and sets its time to t_0, then the time delta between each sensor is given as $= t_2 - t_1 = (t_0 + 200) - (t_0 + 100) = 100$ ms.

Protocols exist to mitigate this variance, however due to the unpredictability in network latency across various environments, this method is deemed unsuitable. Rather, it is preferential to synchronise each device across local, controllable networks using a client-server methodology while applying such mitigating protocols, thereby minimising the variance introduced by external variables such as network speed.

Network Time Protocol (NTP) is one such mitigating protocol for clock synchronisation between participating systems over variable-latency networks. It is one of the oldest protocols in use, being in operation since 1985. The typical accuracy of NTP coordinated systems over Internet-based networks ranges from 5 ms to 100 ms [17], and demands better than 128 ms time deltas. In a rough implementation of NTP, a client will synchronise its clock with a remote server by computing the round-trip delay,

$$\delta = (t_3 - t_0) - (t_2 - t_1) \tag{1}$$

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \tag{2}$$

where δ is the round-trip delay, θ is the estimated offset between clocks, t_0 is the client's timestamp upon transmission of the clock sync request, t_2 is the server's timestamp upon transmission of the clock sync response and t_3 is the client's timestamp upon receiving the clock sync response.

The most accurate offset, θ_0 across multiple round-trips is measured at the smallest yielded delay, δ_0 [18, 19]. Typically eight measurements are performed.

The complete NTP specification will also perform statistical analysis to discard outliers and estimate the best time from networked devices, in order to best correspond each connected device to Coordinated Universal Time (UTC). In the application of synchronising two devices in a client-server methodology, this coordination with global clocks is treated as unnecessary, and particular focus is only paid to the time delta between each participating device.

It is worth noting that Bluetooth provides time-synchronisation across Bluetooth Piconets and Bluetooth Scatternets via the internal clock that is maintained to control packet transmission and reception, however does not provide this service externally to applications. Existing work demonstrates the ability to implement time synchronisation across Bluetooth Scatternets via the internal clock to the accuracy of a few milliseconds [10], as discussed in subsection 3.2. This approach relies upon access to the Bluetooth API to gain limited access to the internal clock, which is not readily available upon the Android platform. Dedicated Bluetooth hardware devices that provide such API functions may be able to implement this synchronisation feature to an accuracy greater than most network-based synchronisation protocols.

3.4 Android

All information below is relevant to Android as of API level 20, i.e. Android 4.4, and can be found via the Android developer documentation [20] and source documentation [21].

Android, based on the Linux kernel, is a mobile operating system (OS) popular with mobile handset companies in smartphones and tablet computers, with growing popularity in televisions, card and wearable devices such as watches. Chiefly developed by Google since August 2005, Android follows the open source Apache License which allows modification and redistribution. It is therefore freely available for use.

The benefit of Android devices as data collection agents and sensors lies in the global availability of budget smartphones equipped with position and inertial motion sensors, and connectivity devices including Bluetooth. An application (app) can request permission to access and leverage these devices by declaring its intention to use required software and hardware components in the app "manifest" file.

For example, in order to utilise an Android platform's Bluetooth adapter, the app must declare the Bluetooth permission `BLUETOOTH`. This is required to perform any Bluetooth communication, such as transferring data. For an app to initiate device discovery to find available Bluetooth devices to connect with, an app must declare the `BLUETOOTH_ADMIN` permission.

The Android platform's support for the Bluetooth network stack exists via the application framework providing the Android Bluetooth API, which in turn gives access to functionality on the local Bluetooth adapter. The Android Bluetooth API allows for scanning for other Bluetooth devices, querying the Bluetooth adapter for paired Bluetooth devices, establishing RFCOMM channels, connecting to other Bluetooth devices, transferring data across established Bluetooth connections, and managing multiple connections (e.g. across a piconet). Note that the API does not provide access to lower-level Bluetooth adapter functionality, such as the Bluetooth clock. The Android Bluetooth API also supports a select number of Bluetooth Profiles, including Headset, A2DP, and Health Devices (HDP).

In order to create a connection between an Android app to an external Bluetooth device, the server must open a server socket and the other device must initiate a connection to this using the server device's MAC address. Once both devices have a connected Bluetooth Socket on the same RFCOMM channel, the devices can begin transmitting streamed data. Each device will have an input stream and an output stream that handles the transmission and reception of data respectively, which can be read from or written to using simple `read[byte[])` and `write(byte[])` function calls.

The Android platform also provides standard access to several sensors including motion (inertial) sensors and position sensors via API calls. The position sensors include geomagnetic rotation vector sensors, geomagnetic field sensors, proximity sensors, and a game rotation sensor. Among the motion sensors provided are two that are hardware-based: the accelerometer and gyroscope; and three that will either be hardware-based or software-based: the gravity, linear acceleration, and rotation vector sensors. In Android, a software sensor is a virtual sensor that obtains its data by combining or translating the data from multiple other sensors into its own distinct data. For example, a software sensor may use the accelerometer and magnetometer on some devices, but on other devices may additionally use the gyroscope to further augment its readings.

Of particular concern is the rotation sensor, which returns a rotation vector across all 3 axes of rotation. It is used for tasks such as detecting gestures, monitoring angular change, and monitoring relative orientation changes. The rotation vector is a continuous-polling sensor that takes a sample rate and relies upon the underlying base accelerometer, gyroscope (as of Android 4.0), and magnetometer sensors. The rotation vector is usually obtained by integration of accelerometer, gyroscope and magnetometer readings for improved stability and performance.

The rotation sensor therefore inherently provides amalgamated sensor data to reduce errors introduced by drift, eliminating the requirement for implementing a Kalman filter or similar. It is important to note this fact, as any implementation developed external to the Android platform will require manual filtering of sensor data to eliminate drift errors. Similarly, devices prior to Android 4.0, or devices that do not include a gyroscopic sensor, running the app may contain inaccuracies. It is suggested therefore to additionally gather and log any other sensor information (including magnetometer and accelerometer data) during application runtime so that this data may later be used to filter the data. The rotation sensor could therefore be considered as a broad-phase, in-place sensor that produces slightly inaccurate data on the spot, more precise results are deferred for later calculation.

The sensor is accessed in Android via `Sensor.TYPE_ROTATION_VECTOR`. Elements of the returned rotation vector are unitless. The reference coordinate system, known as the world frame, follows the East-North-Up coordinate frame defined as a direct orthonormal basis where the X axis "is tangential to the ground at the device's current location and points approximately East", the Y axis "is tangential to the ground at the device's current location and points toward the geomagnetic North Pole", and the Z axis "points toward the sky and is perpendicular to the ground plane" [20]. The device orientation within the global frame is defined as an orientation vector x, y, and z. The rotation vector is encoded as an array of the four reordered components of a unit quaternion:

$$\left[\texttt{rot\_axis.x} * \sin\left(\frac{\theta}{2}\right), \texttt{rot\_axis.y} * \sin\left(\frac{\theta}{2}\right), \texttt{rot\_axis.z} * \sin\left(\frac{\theta}{2}\right), \cos\left(\frac{\theta}{2}\right)\right]$$

Where `rot_axis.(x,y,z)` are the world frame coordinates of a unit length vector that represent the rotation axis, and θ is the angle of rotation. The rotation sensor also provides a fifth element, specifying the estimated accuracy in radians of the rotation vector. The rotation can be seen as rotating the phone by an angle around an axis `rot_axis.(x,y,z)` to go from the world reference device orientation to the current device orientation.

4 Angle Estimation and Orientation Formalisms

Orientation (often referred to as angular position, or attitude), which describes the direction that an object is facing, together with position, which describes where an object is positioned, are used to fully describe the placement of an object in the space in which it is located in. Typically this space would refer to a 3-dimensional Euclidean vector space (as commonly described by standard Cartesian coordinate systems) which is an orthonormal basis, and can be referred to as the world reference frame.

4.1 Background

It is important to note the difference between orientation and rotation of an object: the former describes a property of an object that gives the direction that the object is facing within the world reference frame, while the latter represents a transformation of the object's orientation relative to the world reference frame. For example, an object facing East refers to the object's orientation. An object facing East that is transformed counter-clockwise to face North undergoes a rotation. Rotations occur around a fixed point of the world reference frame, and so any object not centred at this point will also have its position translated under a rotation. In the context of the problem domain, we may assume each object is always centred at its point of rotation, as object position can be discarded without loss of precision. Due to this, we may represent the orientation of any object as a rotation around the origin of its reference frame, or alternatively we may represent an object orientation as a rotation *of the reference frame.*

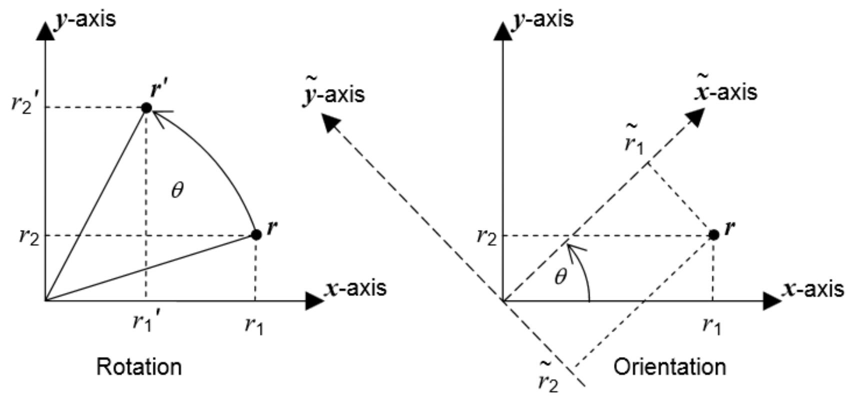


Figure 4: Rotations (left) and Orientations (right), expressed as a rotation around a frame and a rotation of a frame respectively.

Source: Technical Concepts: Orientation, Rotation, Velocity and Acceleration, and the SRM [22]

This section will briefly introduce the mathematical and geometric concepts of coordinate systems and bases as reference frames. It will then discuss representations of rotations in 3 dimensions, and finally join these concepts in order to analyse how to transform rotational data from the problem domain into usable geometric constructs and vice versa.

4.1.1 Fields, Vector Spaces, and Bases

We will begin by loosely introducing the concept of fields, vector spaces, and bases:

- A given field, F, is any set of two or more elements that conform to the field axioms for addition and multiplication–those of associativity, commutativity, distributivity, identity, and inverses–and are commutative division algebra. Examples include the real numbers, $\mathbb{R}$, complex numbers, $\mathbb{C}$, and rational numbers, $\mathbb{Q}$.
- A vector space V over a field F is a set of n-dimensional vectors that is closed under vector addition and scalar multiplication, where the vectors in V are composed of the scalars in the field F. The most basic example of a vector space is n-dimensional Euclidean space, $\mathbb{R}^n$, where every element is represented by a list of n real numbers.
- A basis is a set of linearly independent vectors that can be used in a linear combination to represent every vector within a given vector space, which then form the axes of the space (as with the x-y-z axes in 3-dimensional Euclidean space). The number of basis vectors defines the dimension of the space.

4.1.2 Frames of Reference within Coordinate Systems

In geometric applications, a frame of reference refers to a coordinate system (or set of axes) used to measure relative properties of objects such as position and rotation. More generally, a reference frame defines a single mathematical geometric space for all moments in time such that multiple objects can be placed and distinguished within it. Position and orientation is therefore dependent on the reference frame, while translations and rotations that transform these properties are relative to the object and need only be applicable to the geometry of the reference frame.

In representing position and orientation properties, the reference frame must be defined as an ordered set of coordinate numbers $p = (x_1, x_2, \ldots, x_n)$ within a coordinate system of a vector space V over a field F (which leads to the definition of the frame's coordinate surfaces and coordinate lines), and a set of basis vectors $B = v_1, \ldots, v_n$ where B must necessarily be a basis of V such that:

- for all $a_1, \ldots, a_n \in F$, if $a_1 v_1, \ldots, a_n v_n = 0$ then $a_1 = \ldots = a_n = 0$, and
- for every $x \in V$, there exists an $a_1, \ldots, a_n \in F$ such that $x = a_1 v_1 + \ldots + a_n v_n$.

These elements together form a frame of a coordinate system. If the basis vectors are orthogonal (that is, mutually perpendicular) to one another at every point, i.e. for all pairs of vectors (v_i, v_j) where $i \neq j$ and $v_i, v_j \in B$ the inner product $\langle v_i, v_j \rangle = 0$, then the coordinate system is an orthogonal coordinate system. If the vectors are orthonormal (that is, have unit length one) such that for each $v_i \in B$, $\langle v_i, v_i \rangle = 1$, then the basis is an orthonormal basis. This is the case for n-dimensional Euclidean space $\mathbb{R}^n$ with standard basis formed by $(e_1, \ldots, e_n)$, where each vector e_i is the vector containing all zeros except for a 1 in the ith coordinate; e.g. $e_1 = (1, 0, \ldots, 0), e_3 = (0, 0, 1, \ldots, 0)$, etc. A basis vector can be thought of as an axis of a space.

Remember that within a vector space, we may define many bases, and hence, many frames of reference. Specifically, we may transform an orientation within one reference frame to another reference frame within the same vector space. Hence, for two orientations a_1, a_2 within a reference frame, if a_1 is composed of basis vectors, we may transform a_2 such that it is expressed *from the reference frame of* a_1. For example, in anatomy, it is more convention to express the orientation of the lower-arm below the elbow *from the reference frame of* the upper-arm above the elbow.

4.1.3 Euclidean Space

Euclidean vector-space is the most traditionally recognised coordinate system and reference frame. 3-dimensional Euclidean space ($\mathbb{R}^3$) specifically is used almost synonymously with the Cartesian coordinate system, wherein all points in space may be expressed as a combination of three coordinates–typically labeled as x, y, and z.

Euclidean space includes the 3-parameter geometric model that most accurately represents the physical universe. All of the orientation formalisms explored in this section will be applicable to Euclidean space.

4.2 Rotation Formalisms in Euclidean Space

With a suitable reference frame in mind, objects can be placed within the reference frame with position and orientation properties (represented by coordinate points), and hence be transformed under operations such as translation and rotation.

4.2.1 Principal Rotations

One of the most conventional methods of representing any given rotation is to express it in terms of a series of principal rotations around each of the axes (basis vectors) of the vector space to achieve a final orientation.

4.2.1.1 Euler Angles

In 3 dimensional Euclidean space, a popular–and easily visualised–method of such rotations are Euler angles. Euler angles are a specification of a rotation obtained by applying three consecutive principal rotations. There are twelve distinct conventions for applying Euler angles, which differ in the order and selection of axes to apply a principal rotation about.

We will take the z-x-z convention as example. Hence, let l be an object in the space represented as a line of nodes, x, y, z be the 3 primary axes, and x', y', z' be the transformed axes after rotation. Then let α be the angle between the x-axis and l, β be the angle between the z-axis and the z' axis, and γ be the angle between l and the x' axis. α, β, and γ are sometimes referred to as the spin, nutation and precession angles respectively. These three angles specify three consecutive principal rotations about the fixed principal axes z and x, where we first rotate around the z-axis by angle α, followed by the x-axis by angle β, then about the z-axis again by angle γ. Combined, this is expressed as $R_z(\gamma)R_x(\beta)R_z(\alpha)$ [22, p. 15].

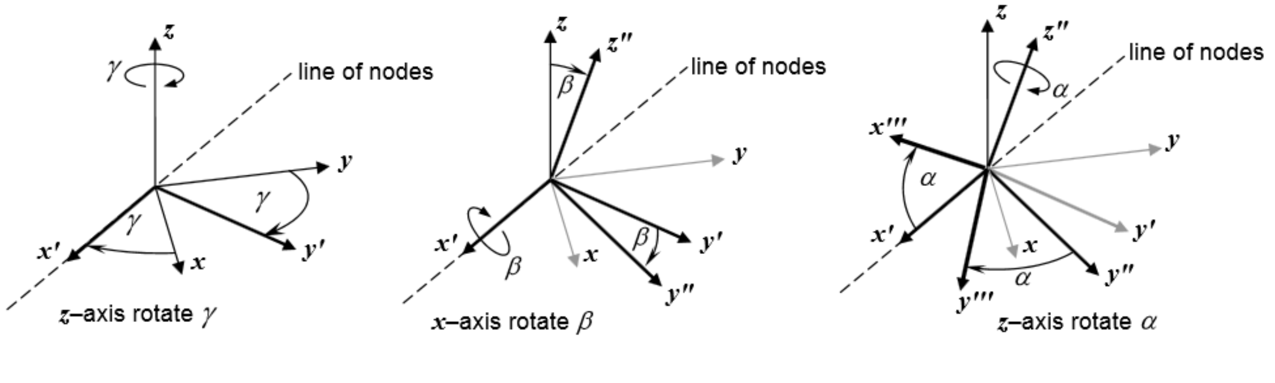


Figure 5: Euler Rotations expressed as three sequential operations
Source: Technical Concepts: Orientation, Rotation, Velocity and Acceleration, and the SRM [22]

4.2.1.2 Tait-Bryan Angles

Another popular method are Euler angles in the x-y-z convention. Taking l, x, y, z and x', y', z' as before, we now define ϕ as the angle between l and y', θ as the angle between z and the

$y'z'$ plane, and ψ as the angle between y and l. The first rotation is then about the x-axis by angle ϕ, then about the y-axis by angle θ, then about the z-axis by angle ψ. This is given as $R_z(\psi)R_y(\theta)R_x(\phi)$. The Euler angles used in this convention are commonly referred to as Tait-Bryan angles, or nautical angles, where ϕ, θ, ψ refer to "roll" about the x-axis, "pitch" about the y-axis, and "yaw" about the z-axis respectively [22, p. 18].

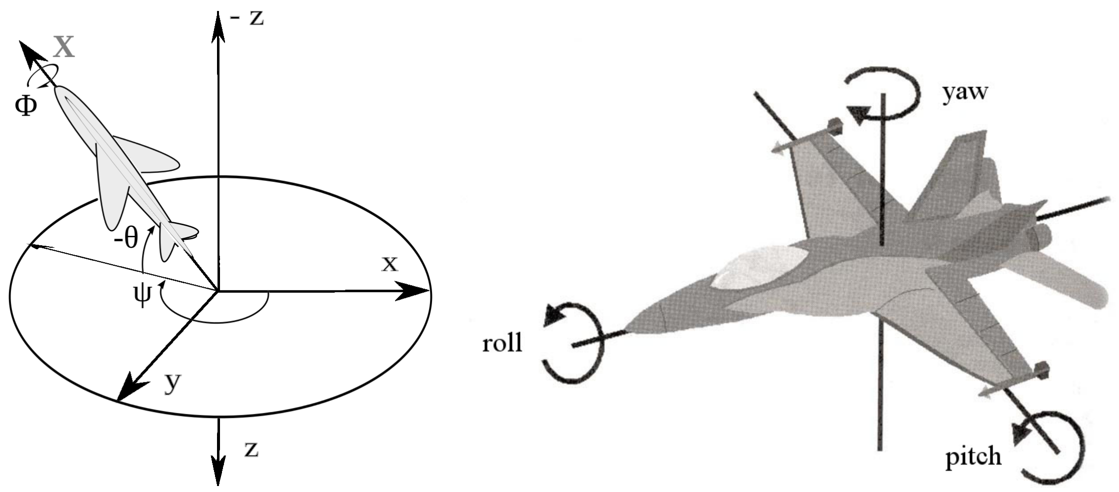


Figure 6: Tait-Bryan Rotations expressing roll, pitch, and yaw
Source: http://en.wikipedia.org/wiki/Euler_angles

4.2.1.3 Rotation Matrices

Other systems of representation include rotation matrices, which is a matrix that is used to perform rotations in Euclidean space. A rotation matrix is an orthogonal matrix with real entries, and determinant 1. More formally, that is $R^T = R^{-1}$ and $\det R = 1$. The values of a rotation matrix represent coordinate rotations relative to a reference frame. A Rotation matrix R can be applied to a coordinate vector v of the same dimension in Euclidean space by a simple matrix multiplication process Rv. As matrix multiplication has no effect on zero vectors, rotation matrices have the limitation that they cannot perform orientation rotations around the origin of the coordinate system [22, p. 18].

4.2.1.4 Gimbal Lock

It is important to highlight the limitations of these systems, so that their use may be limited. Representations such as Tait-Bryan are intrinsic to the problem domain, however performing operations with these may introduce gimbal locking errors. Gimbal lock refers to the loss of degrees of freedom in a rotational system when two of the axes of rotation are made parallel.

This is made obvious in Euler angle formalisms, particular Tait-Bryan mechanisms which can be seen to refer to a gyroscope mounted in three nested gimbals each corresponding to roll, pitch, and yaw. In such a scheme, there exist critical angles for the middle gimbal that restrict the rotational degrees of freedom from three to two when these angles place the axes of two of the gimbals into a parallel configuration. To illustrate, in the case of the Euler z-x-z convention, we take

$$\beta = n\pi \mod 2\pi = \begin{cases} 0, & \text{if } n = 0, 2\pi, \dots \\ \pi, & \text{if } n = \pi, 3\pi, \dots \end{cases}$$

such that the xy-plane and the $x'y'$-plane intersect on l. Then the rotation becomes

$$\begin{cases} \beta = 0: & R_z(\gamma)R_x(0)R_z(\alpha) = R_z(\gamma)R_z(\alpha) = R_z(\gamma+\alpha) \\ \beta = \pi: & R_z(\gamma)R_x(\pi)R_z(\alpha) = R_z(\gamma)R_z(-\alpha) = R_z(\gamma-\alpha) \end{cases}.$$

It is best to use principal rotation mechanisms only for conversion between human readable representations and more functional representations, which we will now introduce.

4.2.2 Angle-Axis Representation

A much more versatile rotation formalism exists in the form of the axis-angle representation, evolved directly from Euler's rotation theorem. The theorem loosely states that for any rotation in Euclidean space, there exists a parametrisation consisting of only two values: a unit vector $\hat{e}$ giving the direction of an axis of rotation, and an angle θ describing the magnitude of the rotation [22, p. 10].

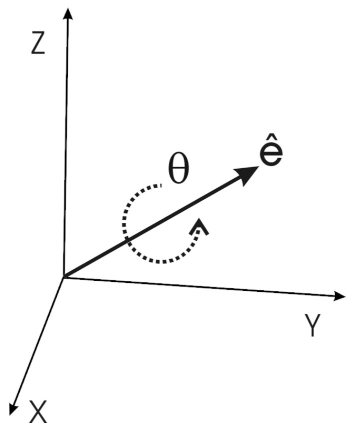


Figure 7: An axis of rotation, $\hat{e}$, and an angle of rotation, θ, expressed in axis-angle representation. *Source: http://en.wikipedia.org/wiki/Axis-angle_representation*

Similarly, for any arbitrarily large sequence of rotations about any given axis within a vector space, there is a single axis-angle parametrisation to represent this transformation. This gives rise to a rotation vector, also known as a Euler vector, which is given as $e = \theta\hat{e}$, i.e. the product of the vector $\hat{e}$ and the scalar θ.

In 3 dimensions, this can be expressed by Equation 3

$$\begin{aligned} \langle \texttt{axis}, \texttt{angle} \rangle = \quad (\hat{e}, \theta) \quad &= \left(\begin{bmatrix} \hat{e}_x \\ \hat{e}_y \\ \hat{e}_z \end{bmatrix}, \theta \right), \text{which gives} \qquad (3) \\ e = \quad \hat{e}\theta \quad &= (\hat{e}_x\theta, \hat{e}_y\theta, \hat{e}_z\theta)\,. \qquad (4) \end{aligned}$$

4.2.3 Quaternions

The quaternions in mathematics are a hypercomplex number system denoted by $\mathbb{H}$, first introduced in 1843 by William Hamilton that apply rather curiously to mechanics in 3-dimensional space, most notably in rotations. Quaternions form a 4-dimensional composition algebra over the real numbers, therefore equal to a four-dimensional vector space. Elements of $\mathbb{H}$ are non-commutative and instead can be operated upon via addition, scalar multiplication, and quaternion multiplication known as the Hamilton product. Similar to complex numbers that may be written $a + ib$ where $i^2 = -1$, quaternions rely upon the axes i, j, k where $i^2 = j^2 = k^2 = ijk = -1$.

A quaternion q comprises two parts, or four terms: the first part includes the first term, which is known as the real part, while the second term includes the remaining three terms, which is known as the imaginary or complex part. Formally, for a quaternion q, this is denoted by $q = e_0 + e_1 i + e_2 j + e_3 k$, where e_0 gives the real part and $\hat{e} = e_1 i + e_2 j + e_3 k$ gives the imaginary part. This is known as the Hamilton form. This may be expressed as an ordered pair of a scalar and a vector, $q = (e_0, \hat{e})$, or as a four-tuple of reals, $q = (e_0, e_1, e_2, e_3)$.

The structure of a quaternion has significance in the application of quaternion algebra, as the complex and real parts must be treated under their own algebraic rules. The non-commutativity of quaternions, shown simply from the identities $ik = k$ and $ji = -k$ which can be deduced from the fact that $i^2 = j^2 = k^2 = ijk = -1$, impacts upon quaternion algebra [22, p. 25-27].

4.2.3.1 Quaternion Operations

The following equations show the common operations in quaternion algebra. Let $p = p_0 + p_1 i + p_2 j + p_3 k$ and $q = q_0 + q_1 i + q_2 j + q_3 k$ be two quaternions, and let t be a scalar. Then

The scalar multiple is

$$\begin{aligned} tq &= tq_0 + tq_1 i + tq_2 j + tq_3 k \\ &= (tq_0, tq) \\ &= (tq_0, tq_1 i, tq_2 j, tq_3 k) \end{aligned} \tag{5}$$

The additive function is

$$\begin{aligned} p + q &= (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k \\ &= (p_0 + q_0, \hat{p} + \hat{q}) \\ &= (p_0 + q_0, p_1 + q_1, p_2 + q_2, p_3 + q_3). \end{aligned} \tag{6}$$

The Hamilton product of two quaternions pq is given by

$$\begin{aligned} pq = &(p_0 q_0 - p_1 q_1 - p_2 q_2 - p_3 q_3) \\ &+ (p_1 q_0 + p_0 q_1 + p_2 q_3 + p_3 q_2)i \\ &+ (p_2 q_0 + p_0 q_2 + p_3 q_1 + p_1 q_3)j \\ &+ (p_3 q_0 + p_0 q_3 + p_1 q_2 + p_2 q_1)k \\ = &\big((p_0 q_0 - \hat{p}\hat{q}), (q_0 \hat{p} + p_0 \hat{q} + \hat{p} \times \hat{q})\big). \end{aligned} \tag{7}$$

The conjugate of a quaternion is given by

$$\begin{aligned} q^* &= q_0 - q_1 i - q_2 j - q_3 k \\ &= (q_0, -\hat{q}) \\ &= (q_0, -q_1, -q_2, -q_3). \end{aligned} \tag{8}$$

The norm of a quaternion is given by the product of a quaternion with its conjugate

$$\begin{aligned} \sqrt{qq^*} &= \sqrt{q^* q} \\ &= \sqrt{q_0^2 + \langle \hat{q}, \hat{q} \rangle} \\ &= \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}. \end{aligned} \tag{9}$$

The modulus of a quaternion is also thus

$$\begin{aligned} |q| &= \sqrt{qq^*} = \sqrt{q^*q} \\ &= \sqrt{q_0^2 + \langle \hat{q}, \hat{q} \rangle} \\ &= \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}. \end{aligned} \tag{10}$$

The multiplicative inverse of a quaternion is given as

$$q^{-1} = \frac{q^*}{qq^*} = \frac{q^*}{|q|^2}. \tag{11}$$

A unit quaternion has a modulus of 1, and therefore has a multiplicative inverse equal to its conjugate

$$qq^* = q^*q = 1 \implies q^{-1} = q^*. \tag{12}$$

4.2.3.2 Quaternion Rotations

It is these unit quaternions that are of particular relevance to representing rotations, in what is known as versors, rotation quaternions, or orientation quaternions (note: as before, here an orientation quaternion represents an absolute orientation from the origin of a reference frame, while rotation quaternions provide a rotation operation). Versors provide a more numerically stable solution than rotation matrices, that avoid gimbal locking and generally require fewer operations to perform a rotation. They function on the premise of Euler's rotation theorem and axis-angle representation by using the real part of a quaternion as a scalar angle θ and the complex part as a unit vector representing a Euler axis of rotation.

For an axis of rotation given by a unit vector

$$u = (\hat{u}_x, \hat{u}_y, \hat{u}_z) = \hat{u}_x i + \hat{u}_y j + \hat{u}_z k$$

where i, j, k represent the three standard Cartesian axes, a rotation through an angle θ can be represented by a quaternion using an extension of Euler's formula

$$q = e^{\frac{\theta}{2}(\hat{u}_x i + \hat{u}_y j + \hat{u}_z k)} = \cos\left(\frac{\theta}{2}\right) + (\hat{u}_x i + \hat{u}_y j + \hat{u}_z k) = \left(\cos\left(\frac{\theta}{2}\right), \hat{u}\sin\left(\frac{\theta}{2}\right)\right).$$

Then the rotation around a point $p = (0, p_x, p_y, p_z) = p_x i + p_y j + p_z k$ with a real part equal to zero (i.e. a Euclidean vector in three-dimensional space) can be given by evaluating the conjugation of p by q using the Hamilton product from Equation 7

$$p' = qpq^{-1} = qpq^{*} \tag{13}$$

where p' is the rotated point.

Multiple sequential rotations can be easily chained together using the non-commutative product of rotation quaternions. If r and q are unit rotation quaternions and p is a point then

$$r(qpq^{-1})r^{-1} = rqpq^{-1}r^{-1} = (rq)p(rq)^{-1} \tag{14}$$

Which is equal to the result of rotating the point p by q and then by r. Hence, two rotation quaternions can be formed into a single rotation quaternion by taking the quaternion multiplicative result from Equation 7

$$q' = rq \tag{15}$$

where q' is the rotation q followed by r.

Similarly, the opposite rotation of a rotation quaternion can be given by the inverse of the quaternion, as

$$q^{-1}(qpq^{-1})q = (q^{-1}q)p(q^{-1}q) = p. \tag{16}$$

4.3 Application to the Problem Domain

This theory forms the foundation required to apply geometric rotation operations within a given reference frame. As discussed in subsection 3.4, Android returns an orientation vector which can be expressed as a unit quaternion representing its absolute orientation aligned to the magnetic world reference. We will assume the use of a 3-dimensional Euclidean-space reference frame with the alignment $(x, y, z) =$ `(East,North,Up)`.

The rotation vector data returned by the Android rotation vector sensor relative to the world reference frame is expressed in the re-ordered quaternion four-tuple form:

$$q = \left[\hat{u}_x \sin\left(\frac{\theta}{2}\right), \hat{u}_y \sin\left(\frac{\theta}{2}\right), \hat{u}_z \sin\left(\frac{\theta}{2}\right), \cos\left(\frac{\theta}{2}\right)\right]$$

where θ is the angle of rotation and $\hat{u}$ is the axis of rotation, and the scalar component $\cos\left(\frac{\theta}{2}\right)$ is only returned as an optional value depending on the device implementation. In the case of the scalar component not being provided, we may calculate the scalar component of the orientation quaternion as

$$q_{scalar} = \sqrt{1 - |\hat{u}|} = \sqrt{1 - \hat{u}_x^2 - \hat{u}_y^2 - \hat{u}_z^2}.$$

As we are utilising two such sensors to gather the orientation of the thigh and shin at a given time slice, we will express the orientation vectors for each of these sensors for a specific time slice in standard quaternion format, denoted by q and r, with angles θ and ψ and rotation axes $\hat{u}$ and $\hat{v}$ respectively

$$\begin{aligned} q = (q_0, q_1, q_2, q_3) &= \left(\cos\left(\frac{\theta}{2}\right), \hat{u}_x \sin\left(\frac{\theta}{2}\right), \hat{u}_y \sin\left(\frac{\theta}{2}\right), \hat{u}_z \sin\left(\frac{\theta}{2}\right)\right), \\ r = (r_0, r_1, r_2, r_3) &= \left(\cos\left(\frac{\psi}{2}\right), \hat{v}_x \sin\left(\frac{\psi}{2}\right), \hat{v}_y \sin\left(\frac{\psi}{2}\right), \hat{v}_z \sin\left(\frac{\psi}{2}\right)\right) \end{aligned}$$

For the sake of completeness, it can also be shown that, from orientation quaternions such as these, we may extrapolate the orientation of either sensor in angle-axis representation by a simple formula

$$\langle \texttt{axis, angle} \rangle = \begin{cases} \left(\begin{bmatrix} \frac{q_1}{\sin(2\arccos(q_0))} \\ \frac{q_2}{\sin(2\arccos(q_0))} \\ \frac{q_3}{\sin(2\arccos(q_0))} \end{bmatrix}, 2\arccos(q_0) \right) & \text{if } \theta \neq 0 \\ \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, 2\arccos(q_0) \right) & \text{if } \theta = 0 \end{cases}. \tag{17}$$

A simpler and generally superior method [23, 22] of conversion can be given by

$$\langle \texttt{axis, angle} \rangle = \left(\frac{\hat{q}}{|\hat{q}|}, 2\arctan\left(|\hat{q}|, q_0\right)\right) = \left(\frac{\hat{q}}{\sqrt{1-q_0^2}}, 2\arctan\left(\sqrt{1-q_0^2}, q_0\right)\right) \tag{18}$$

However, as we are primarily concerned with the orientation of the second sensor r on the shin relative to the first sensor q on the thigh, we transform the two orientation quaternions to produce a single orientation quaternion q' that gives the orientation of r from the reference frame of q. This is done by taking the inverse of q and taking the quaternion multiplicative of r

$$q' = q^{-1} r = q^* r \tag{19}$$

Hence q' represents the transformation that will rotate from orientation q to orientation r as discussed in subsubsection 4.1.3.

From q' we may calculate the angular distance of the rotation between q and r by solving

$$\theta = 2\arccos(q'_0) = 2\arctan 2(|\hat{q'}|, q_0) \tag{20}$$

This is shown in Figure 8 below where, if we visualise q, r, and q' in axis-angle format, the θ component of q' gives the angular distance between q and r:

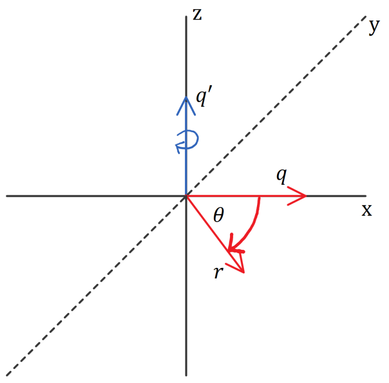


Figure 8: A Difference Quaternion, q', expressing the angular distance between two orientation vectors q and r.

We may convert q' to Tait-Bryan angles and give the angle of rotation of the second sensor relative to the primary sensor, in angles respective to the world reference x-y-z frame [22, p. 39]. As these are given in pitch, yaw, and roll, there is a one-to-one conversion to flexion/extension, valgus/varus, and medial/lateral motion. The formulae are given.

$$\phi = \arctan 2\left((q'_2q'_3 + q'_0q'_1), \frac{1}{2} - (q'^2_1 + q'^2_2)\right) \tag{21}$$

$$\theta = \arcsin\Big(-2(q'_1q'_3 - q'_0q'_2)\Big) \quad \text{with principal values } -\pi/2 < \theta < \pi/2 \tag{22}$$

$$\psi = \arctan 2\left((q'_1q'_2 + q'_0q'_3), \frac{1}{2} - (q'^2_2 + q'^2_3)\right) \tag{23}$$

Further, we may also bypass the requirement to take the difference quaternion and directly take the angle subtended by q and r by simply calculating

$$\theta_d = 2\arccos(q \cdot r) = 2\arccos(q_0r_0 + q_1r_1 + q_2r_2 + q_3r_3) \tag{24}$$

Again, as a consequence of the fact that we may express both quaternions in axis-angle format, the distance θ_d here is equivalent to the flexion/extension (or pitch) angle between the primary and secondary sensor's orientations. This is because the vector orientations of each quaternion, or the axes in axis-angle format–irrespective of the world reference frame–form a natural two-dimensional plane with a single angle, θ_d, between them. This angle is equivalent to the flexion/extension angle of a joint.

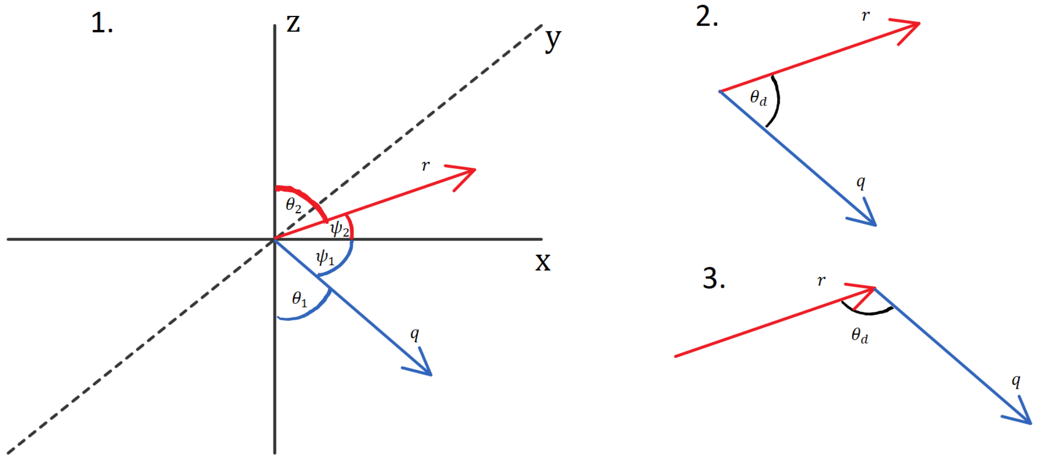


Figure 9: Arc-Distance between Orientations. (1) Shows two orientation vectors with respect to a global reference frame, (2) shows the two vectors relative to each other forming a natural two-dimension vector space, and (3) demonstrates how these vectors can be expressed locally as a flexion/extension angular distance.

However, using this approach we cannot also determine the angle of rotation in any other axis, such as medial/lateral or varus/valgus rotation. It is therefore worthwhile to compute the difference quaternion for later use. However it is worth noting that, in attempting to produce the rotations across each axis (medial/lateral, varus/valgus, flexion/extension) by converting to a formalism such as Tait-Bryan, the resultant rotations may be difficult to work with due to the convention used. As every rotation can be expressed as a combination of many possible rotations, the mathematically produced result is not guaranteed to align with real-world rotations. The topic requires further examination.

5 System Specification

This section covers the analysis of the system requirements, and provides the foundation for the design and implementation of the system continued in section 6.

5.1 Required Functionality

The required functionality defines the goals of the system at a high level. These are given:

- Two sensor nodes connected wirelessly must be able to collect and communicate data in order to determine relative flexional angular difference between their two respective orientations at regular time intervals.
- The sensor nodes must issue an alert if the flexional angular difference exceeds some predefined threshold.
- The collected data must be persisted on an online server either at the end of collection or over time.
- The collected data for each collection session must be able to be analysed either on the sensor nodes immediately after the end of collection, or via an online server.

5.2 Component Specification

To facilitate the identified design requirements, the implementation of the system requires a disjoint separation of components. These components will be described in greater detail throughout this document. The disjunction follows the commonplace client-server paradigm, with the client fulfilling the role of a data collection agent, and the server offering centralised data storage and operations.

5.2.1 Client Sensor Network

The Client Sensor Network handles collection of session data, as well as basic controls for specifying variables in the session, such as sample rates, which sensors to sample from, etc. It is composed of the below components.

5.2.1.1 Master Node

- Initiates the data collection process with an attached slave (see 5.2.1.2)
- Allows calibration and configuration of the data collection session
- Allows collection of time-stamped local sensor data
- Allows time-synchronised aggregation of local sensor data with retrieved sensor data from the slave node
- Performs domain-specific transformations and calculations over aggregated data
- Provides a mechanism to communicate aggregated data with an online data persistence engine (see 5.2.2)

5.2.1.2 Slave Node

- Receives initiation requests from master nodes to begin or end data collection
- Allows collection of time-stamped local sensor data
- Allows communication of local data with the master node

5.2.2 Web Application Server

The Web Application Server handles the data persistence and analysis requirements of the system, and provides a web interface for viewing the data in a presentational format. It is composed of the below components.

5.2.2.1 Data Persistence and Analysis Engine

- Provides an API to receive data packets from connected collection agents and persist it to a data storage engine
- Provides an API to allow clients to retrieve persisted data for analysis
- Provides an API to retrieve, analyse, and transform persisted sensor data as necessary

5.2.2.2 Data Reviewal System

- Provides a visual interface to quickly access persisted data from the data persistence and analysis engine (see 5.2.2.1)
- Provides presentational views of the persisted data such as graphs and tables for simple analysis by medical professionals

6 System Design and Implementation

This section continues the work established in section 5, and further discusses the implementation-specific details of the system.

6.1 Discussion

The Client Sensor Network subsystem is designed using the principles of mobile and sensor computing. It therefore specifies a highly generic and reusable framework for data processing tasks such as aggregation of data between multiple sensors to a master node, and transformations of data fields to produce usable outputs.

Addressing the targeted domain of the study, we build an Android application on top of the data-processing framework as an interface to configure variables, track data, and interact with the Data Persistence and Analysis Engine.

The data-processing framework and accompanying Android application are therefore written using pure Java with minimal use of external libraries or frameworks, except for integration between the Android framework and underlying API.

The web application server is designed with reactive programming principles in mind–that is, the system is highly data-driven and data-oriented, and is therefore designed to be reactive and dynamic as data changes. To this end, the server combines the Node.js platform and Express js framework to produce a fast and scalable web solution targeted to data-intensive real-time applications. The web application server defines an API to process requests for storing and retrieving sensor data, that may be reusable across many different sensor network client applications.

Supporting this is the MongoDB document-oriented, NoSQL database management system for handling data persistence and retrieval. MongoDB uses JSON-like syntax and dynamic schemas to support the integration of data from many different fields. It is therefore highly scalable as data needs increase, and is fully supportive of parallelisable, distributed datasets.

The design of these systems is given in detail through this section.

6.2 High Level System Architecture

The system architecture gives the structure of the complete system. At a high level, this is follows a typical client-server architectural design paradigm, with elements of ubiquitous computing concepts. In this paradigm, the system contains multiple client subsystems connecting to a central server system. This is shown in Figure 10 below.

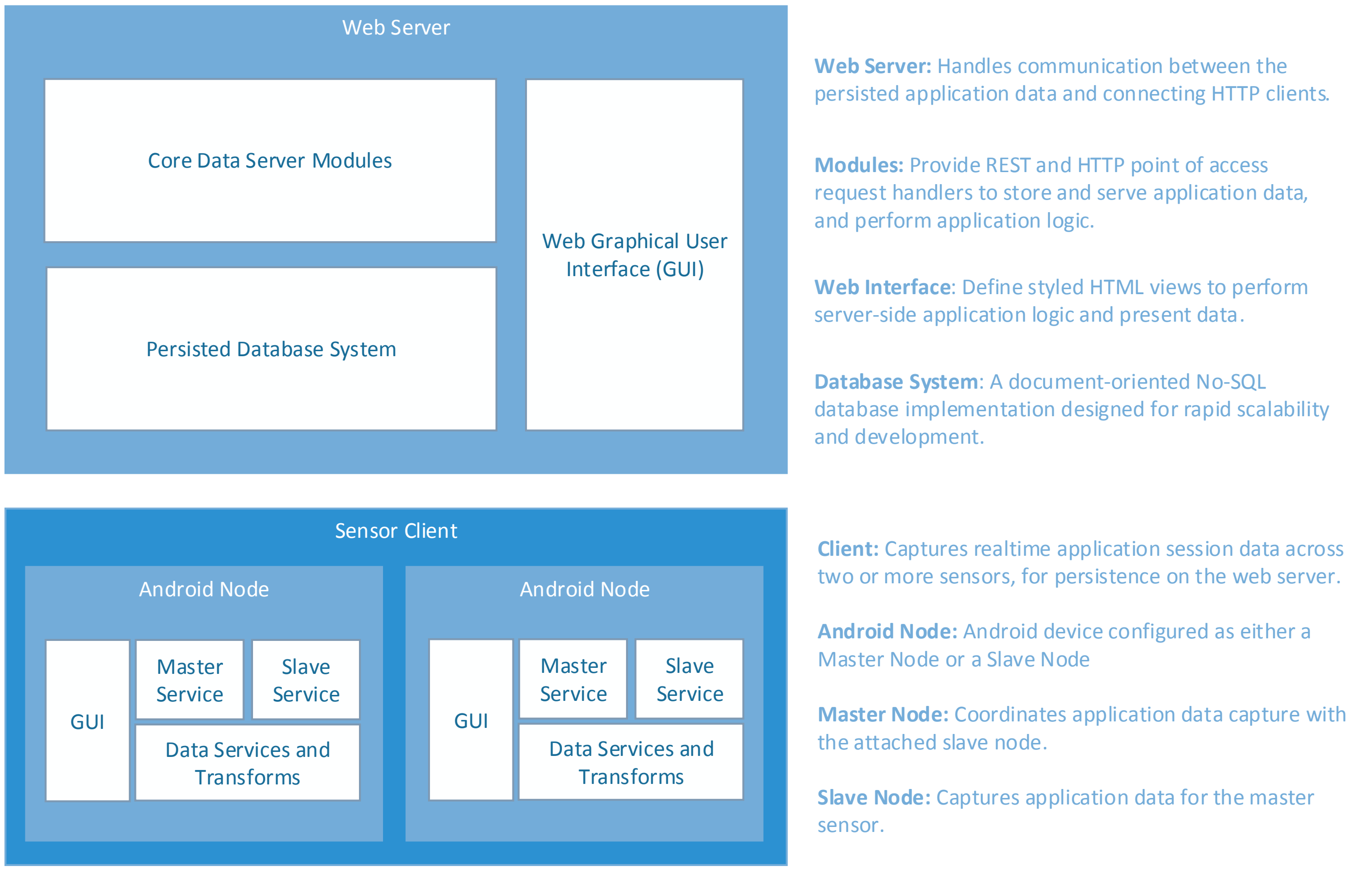

Web Server
Core Data Server Modules
Persisted Database System
Web Graphical User Interface (GUI)
Web Server: Handles communication between the persisted application data and connecting HTTP clients.
Modules: Provide REST and HTTP point of access request handlers to store and serve application data, and perform application logic.
Web Interface: Define styled HTML views to perform server-side application logic and present data.
Database System: A document-oriented No-SQL database implementation designed for rapid scalability and development.
Sensor Client
Android Node
GUI
Master Service
Slave Service
Data Services and Transforms
Android Node
GUI
Master Service
Slave Service
Data Services and Transforms
Client: Captures realtime application session data across two or more sensors, for persistence on the web server.
Android Node: Android device configured as either a Master Node or a Slave Node
Master Node: Coordinates application data capture with the attached slave node.
Slave Node: Captures application data for the master sensor.

Figure 10: High Level System Architecture Diagram

Here, each instance of the client subsystem is actually a discrete mobile sensor network comprised of at least two sensor nodes in a master-slave configuration, as discussed in subsubsection 6.6.1. These sensor networks may be seen as a single entity by the server-side of the system, as they aggregate their individual data at the master node before communicating with the server.

The server subsystem, in accordance with modern reactive techniques and data scalability, provides a data persistence layer accessible via RESTful services to any connected client (n.b. here, client refers to any web client connecting via HTTP), therefore providing the basis for cloud-based data availability and analytics. The subsystem also provides a sample web interface on top of the data persistence layer, available to all modern web browsers. This is discussed further in subsubsection 6.6.2.

6.3 System Data Flow

The primary function of the system specified in subsection 5.1 is data collection, processing, manipulation and transportation. The movement and manipulation of data data within the system is outlined at a high-level in figure Figure 11. More detailed design of the involved system components is discussed in subsection 6.6.

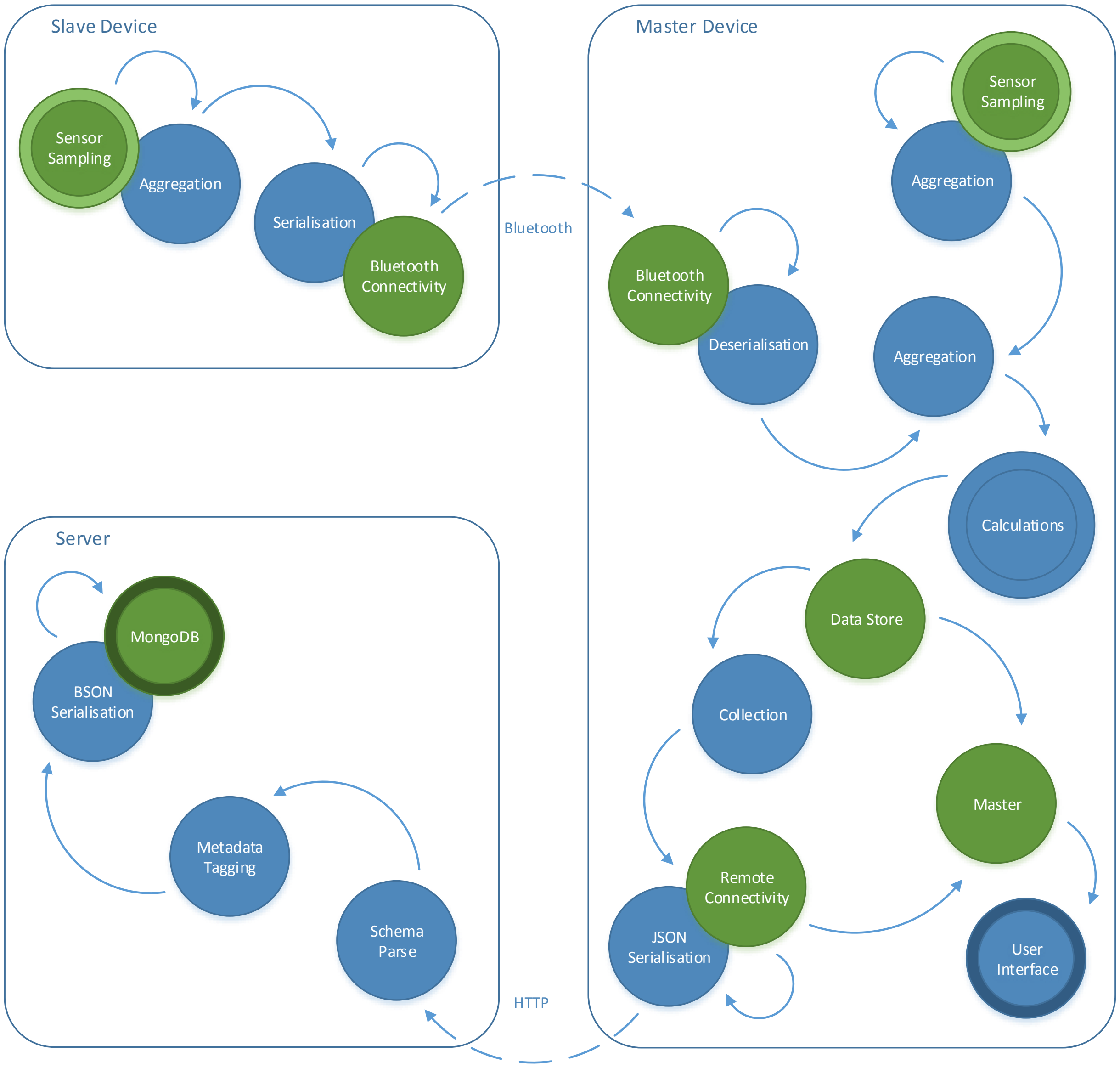


Figure 11: System Data Flow

6.4 Use Case Specification

The use cases of the system represent the atomic operations that each component of the system is required to implement in order to achieve the high level functionality requirements specified in section 5. The use cases of the system are listed in brief below, grouped by the component they belong to.

6.4.1 Client Sensor Network

Interaction with the system for the purposes of collecting data is achieved through the client sensor network. The operations of each node are described below in the corresponding paragraphs.

6.4.1.1 Master Node

The following use cases specify the interactions that the user can perform with the master client devices.

UC #	1.01
Trigger/Goal	Initiate connection
Actor	Physiotherapist
Precondition	Master app running, slave device prepared for connection
Main Flow	1. Configure session collection sensors 2. Configure session critical events 3. Enter user ID for session 4. Select device running slave application 5. Connect to remote server via internet 6. Connect to slave device via wireless protocol 7. Calculate device time offset (time synchronisation)
Extension	1a. No slave found or slave device not available 1. Display error 5a. Web URL unreachable or does not implement required service 1. Display error
Postcondition	Client network connection established

UC #	1.02
Trigger/Goal	Initiate sampling
Actor	Physiotherapist
Precondition	Connection established, slave and remote server available
Main Flow	1. Prepare data pipeline components 2. Initiate master sensors 3. Signal slave device to begin sampling
Extension	3a. Slave device not responding or connection lost 1. Close connection and return to disconnected state 2. Display error
Postcondition	The master and slave are collecting and transmitting data

UC #	1.03
Trigger/Goal	Terminate connection
Actor	Physiotherapist
Precondition	Connection established, slave responding
Main Flow	1. Stop master sensor sampling 2. Signal slave device of connection termination 3. Send final data to server 4. Close wireless connection 5. End session
Extension	2a. Connection to slave device lost or closed 1. Ignore, do nothing
Postcondition	The master and slave have returned to disconnected state

UC #	1.04
Trigger/Goal	Run collection session
Actor	System
Precondition	Master app running, sampling active
Main Flow	1. Receive slave sensor data 2. Receive master sensor data 3. Aggregate sensor data 4. Perform data processing 5. Cache final data 6. Add final data to collection 7. Display calculated knee angle on user interface
Extension	1a. Connection to slave device lost or closed 1. Perform main flow of UC 1.03 6a. If collection is full 1. Send completed data packet to remote server 7a. If calculated angle exceeds defined threshold 1. Generate warning through user interface
Postcondition	Collection session continues

6.4.1.2 Slave Node

The following use cases specify the interactions that the user can perform with the slave client device.

UC #	1.05
Trigger/Goal	Prepare for connection
Actor	Physiotherapist
Precondition	Slave app running
Main Flow	1. Open wireless connection
Extension	1a. Wireless connection unavailable 1. Return to disconnected state 2. Display error
Postcondition	Wireless connection prepared for connection with master

UC #	1.06
Trigger/Goal	Receive start sampling action
Actor	System
Precondition	Connection established with master service
Main Flow	1. Prepare data pipeline components 2. Initiate slave sensors
Extension	-
Postcondition	The slave is collection and transmitting data

UC #	1.07
Trigger/Goal	Terminate connection
Actor	Physiotherapist
Precondition	Connection established with master
Main Flow	1. Stop slave sensor sampling 2. Signal master device of connection termination 3. Close wireless connection 4. End session
Extension	2a. Connection to master device lost or closed 1. Ignore, do nothing
Postcondition	Slave has returned to disconnected state

UC #	1.08
Trigger/Goal	Run collection session
Actor	System
Precondition	Slave app running, sampling active
Main Flow	1. Collect sensor data 2. Aggregate sensor data 3. Send sensor data over wireless connection
Extension	1a. Connection to master device lost or closed 1. Perform main flow of UC 1.06
Postcondition	Collection session continues

UC #	1.09
Trigger/Goal	Receive stop sampling action
Actor	System
Precondition	Connection established with master, sampling active
Main Flow	1. Terminate slave sensor sampling
Extension	-
Postcondition	The slave is not collection sensor data

6.4.2 Web Application Server

Interactions with the Web Application Server support the ability to view, analyse, and manage the sensor data that has been collected using the client sensor network. These operations are divided between two logical components, and are detailed in corresponding paragraphs below.

6.4.2.1 Data Persistence and Analysis Engine

The following use cases specify the interactions that the user can perform with the web application server data persistence and analysis engine.

UC #	2.01
Trigger/Goal	Store session data
Actor	Web Client, System
Precondition	A web client has sent session data to the web application server
Main Flow	1. Retrieve and parse session data 2. Transform the data to save to database 3. Insert the session data to database 4. Respond with success to web client
Extension	1a. Error retrieving data from client 1. Respond with error to web client 3a. Error connecting to or storing data to database 1. Respond with error to web client
Postcondition	The session data is persisted

UC #	2.02
Trigger/Goal	Retrieve list of session data
Actor	Web Client, System
Precondition	A web client has requested a list of all existing sessions
Main Flow	1. Retrieve session data from database 2. Transform data to remove unnecessary fields or duplicates 3. Transform data to correct response output format 4. Respond to client with transformed data
Extension	1a. Error retrieving data from database 1. Respond with error to client
Postcondition	A list of sessions is returned to the client

UC #	2.03
Trigger/Goal	Retrieve session details
Actor	Web Client, System
Precondition	A web client has requested all of a session's data
Main Flow	1. Parse the request and retrieve the session id 2. Retrieve the session data from database 3. Transform the data to calculate any missing fields 4. Transform the data to a response output format 5. Respond with data to client
Extension	2a. Error retrieving data from database 1. Respond with error to client
Postcondition	The session data is returned to the client

UC #	2.04
Trigger/Goal	Delete a session
Actor	Web Client, System
Precondition	A web client has requested a session be deleted
Main Flow	1. Parse the request and retrieve the session id 2. Delete the session data from database 3. Respond with success to client
Extension	2a. Error deleting data from database 1. Respond with error to client
Postcondition	The specified session data is removed from the database

6.4.2.2 Data Reviewal System

The following use cases specify the interactions that the user can perform with the web application data reviewal system.

UC #	2.05
Trigger/Goal	View all sessions
Actor	User, System
Precondition	A user has requested to view a list of all existing sessions
Main Flow	1. Create user interface 2. Retrieve session data (see UC 2.02) 3. Format data into user interface 4. Display interface to user
Extension	3a. Error returned from retrieving session data 1. Format error data into user interface
Postcondition	A list of sessions is shown to the user

UC #	2.06
Trigger/Goal	View a session
Actor	User, System
Precondition	A user has requested to view a session's data
Main Flow	1. Create user interface 2. Retrieve the session data (see UC 2.03) 3. Format data into user interface 4. Display interface to user
Extension	3a. Error returned from retrieving session data 1. Format error data into user interface
Postcondition	The session is shown to the user

UC #	2.07
Trigger/Goal	Delete a session
Actor	User, System
Precondition	A user has requested to delete a session
Main Flow	1. Execute the request (see UC 2.04) 2. Refresh the current view
Extension	1a. Error deleting data from database 1. Halt
Postcondition	The specified session is deleted and can no longer be accessed

6.5 Data Analysis and Design

The design of the system is primarily centred around its data and associated transformations. This results in a system that is highly driven by its data requirements and the produced data analysis, in keeping with the principles of reactive programming. Data analysis hence consists of a basic Entity-Relation (ER) analysis to identify the primary and secondary data objects (entities) within the scope of the system in a relational context. These entities are shown to chiefly centre on the concept of a “session”, containing timeseries datapoints and event constraints. The timeseries datapoints represent the client sensor network’s data, such as orientation, at a discrete point in time, while the event constraints indicate the safe and/or critical thresholds of data within the timeseries datapoints.

A tertiary entity, a “user”, is introduced as a means to logically group sessions by whom they apply to. This relational approach provides a good first step at analysing the requirements of the system, as well as providing an effective schema for storing data on the sensor nodes using SQLite, as discussed in subsubsection 6.6.1.

In this model, the system’s schema is centric around capturing sensor data at particular time intervals, and sorting this data into an encompassing session. These sessions may then be loosely grouped by the user they apply to. In ER notation, this would be described as: “A user has many sessions, where a session has many sensor data points and many event constraints. Conversely, a constraint and data point has only one session, and a session has only one user.” This relation is shown in Figure 12.

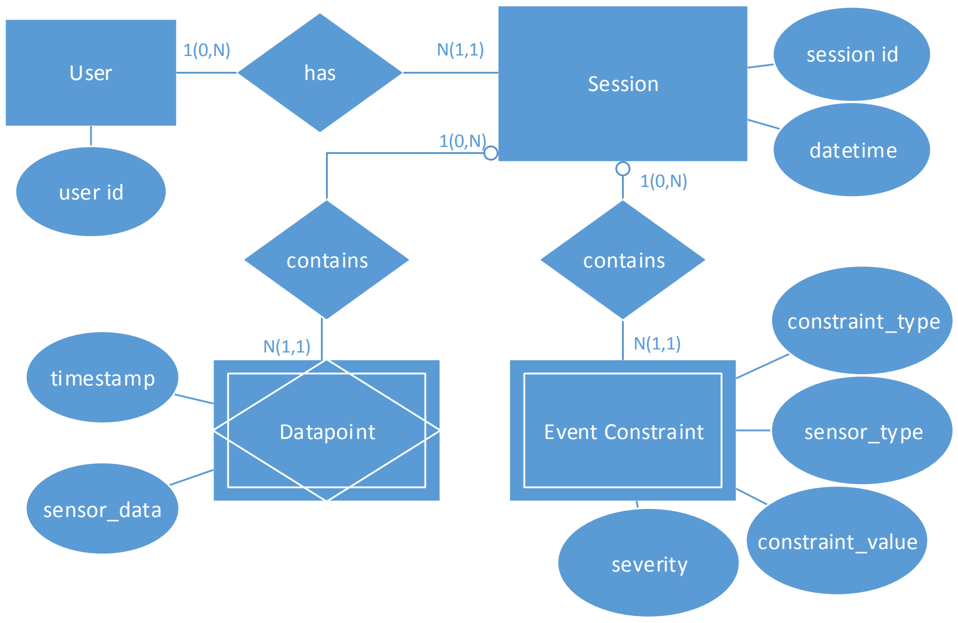
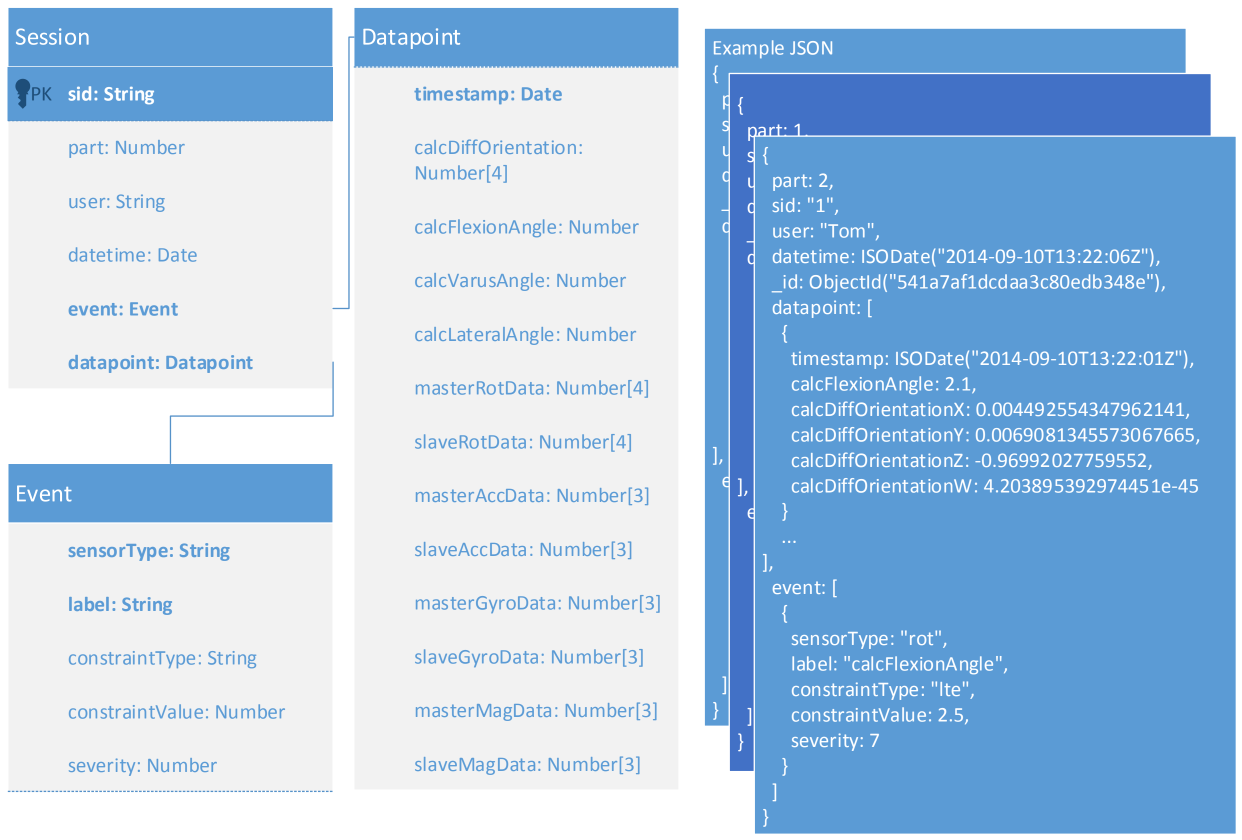


Figure 12: Entity-Relationship Data Model

The schema is then denormalised and given as a document-oriented schema, for use in NoSQL databases with dynamic schemas. Due to the nature of the system operating, first and foremost, as a data collection and storage agent, a NoSQL-style database is advantageous for handling the high volumes of timeseries data required to be persisted. The denormalisation process yields a document-oriented schema comprising a single document-type; a session with both datapoints and events embedded as subdocuments. This is shown in Figure 13. Note that the "part" field is introduced so that a session may be split into multiple documents, each containing a subset of the timeseries datapoints, that would then need to be grouped chronologically. Hence, each session document represents a contiguous "chunk" of a complete session, which allows the sensor client to dump discrete packets of arbitrarily-large timeseries data to the data store as separate documents. These documents may then later be aggregated to yield the complete session.



```
{
  part: 2,
  sid: "1",
  user: "Tom",
  datetime: ISODate("2014-09-10T13:22:06Z"),
  _id: ObjectId("541a7af1dcdaa3c80edb348e"),
  datapoint: [
    {
      timestamp: ISODate("2014-09-10T13:22:01Z"),
      calcFlexionAngle: 2.1,
      calcDiffOrientationX: 0.004492554347962141,
      calcDiffOrientationY: 0.0069081345573067665,
      calcDiffOrientationZ: -0.96992027759552,
      calcDiffOrientationW: 4.203895392974451e-45
    }
    ...
  ],
  event: [
    {
      sensorType: "rot",
      label: "calcFlexionAngle",
      constraintType: "lte",
      constraintValue: 2.5,
      severity: 7
    }
  ]
}
```

Figure 13: ER Data Model

Note that this schema design allows for chunked data retrieval when serving web clients without the server being required to perform any additional work to partition the data. That is, a web page may request the complete session in separately contiguous parts, in order to limit data usage. This is made particularly efficient by the client sensor network persisting data in fixed-size "chunks". Further, the document-oriented style of this schema allows for Big Data techniques to be easily applied to the persisted data, as discussed in section 8.

The database management system chosen for managing this data is MongoDB, a document-oriented, NoSQL data storage engine that uses the BSON (Binary JSON) format. The implementation uses a free-tier MongoDB "sandbox" instance running on Compose, a fully-managed platform used by developers to deploy, host and scale databases, that allows 512 MB of SSD-backed MongoDB storage in the cloud.

6.6 Component Design

We now present the individual design of each component identified in subsection 5.2.

6.6.1 Client Sensor Network

As has been discussed, the specification for the system's client network given in subsection 5.2 outlines a data processing pipeline. This pipeline is further defined as a data flow diagram in subsection 6.3, with the following characteristics:

- Sensor data forms the source input of the pipeline
- Discrete operations, such as domain-specific calculations, format manipulation, and transportation, form the pipeline processes
- Server storage and use by the user interface form the final destinations and states of the pipeline

Analysis of this data flow shows that each process involved can be indivisibly encapsulated, and the internal processes can be defined in isolation from the remainder of the flow. This simple principle echoes the principles of modular design, which is the foundation of the client sensor network design.

The client network comprises a layered architecture, which is illustrated by Figure 14. These layers are explored in sections below, but in brief the architecture can be summarised as the two overarching layers indicated on the left-hand side of Figure 14.

The lower layer defines a generic data processing framework. This layer provides a framework for defining atomic data-processing components that operate on a flexible data structure. This framework interfaces with the Android OS to allow for operations such as performing data processing in the background and on dedicated threads. This layer extends to include a collection of reusable data-processing components defined using the framework to perform basic, ubiquitous tasks.

The higher layer wraps the data-processing framework in domain-specific logic to build the desired data flow (subsection 6.3), and introduces user interfaces to form an interactive application. This higher layer includes the definition of specialised data-processing components for domain-specific calculations and coordination of all involved data components.

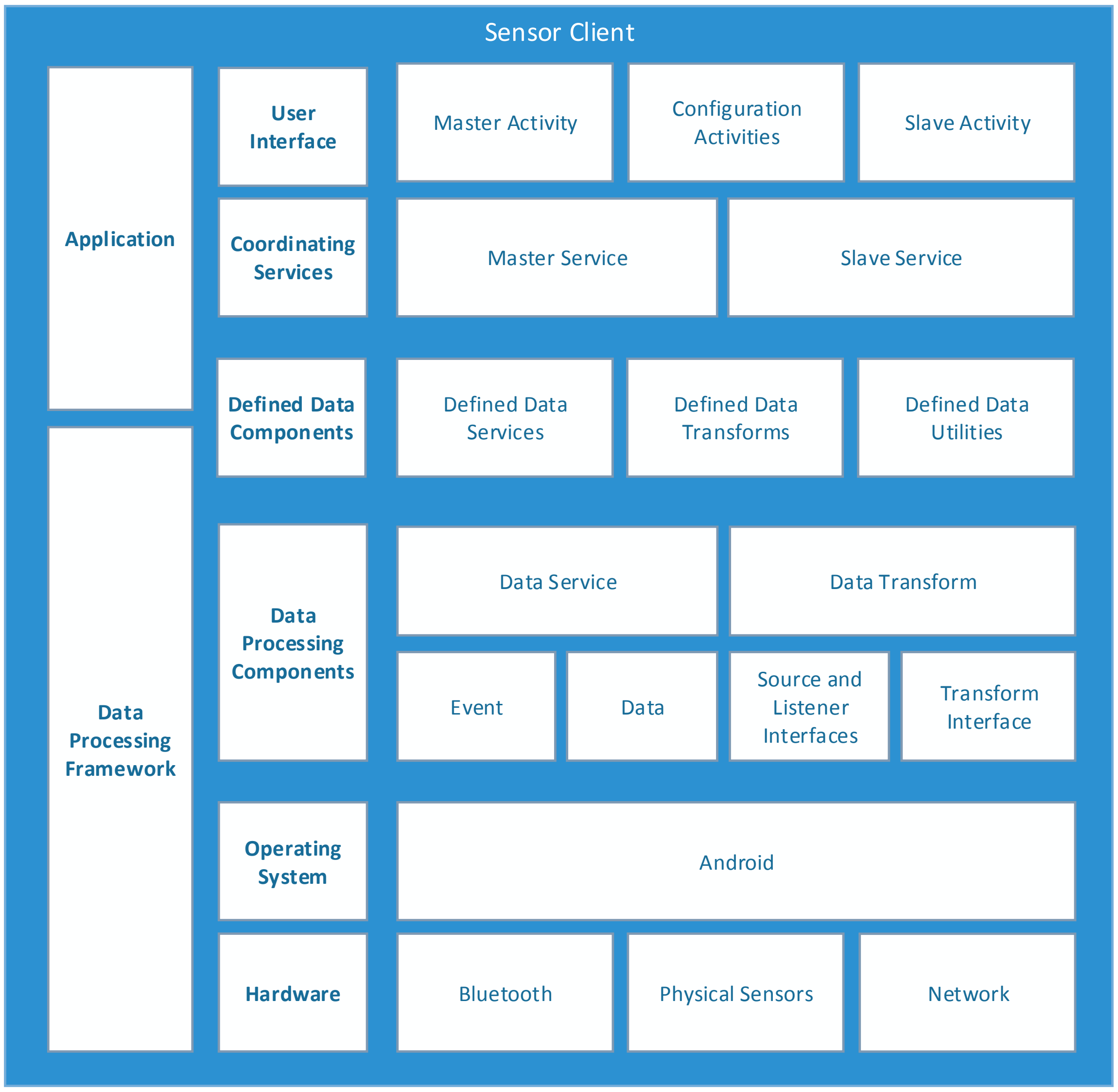


Figure 14: Layered Architecture of Client Node

6.6.1.1 Android Operating System and Hardware

The relevant benefits and advantages of the Android OS and its relation to device hardware is detailed in subsection 3.4.

6.6.1.2 Data-Processing Components

The data-processing framework revolves around the classes defined in Figure 15. At the lowest level, processing involves data being stored and represented by instances of the `Data` class, while implementations of `IDataListener`, `IDataSource` and `IDataTransform` are responsible for transportation and manipulation of `Data` objects. The classes `DataTransform` and `DataService` are implementations of the above interfaces with embedded logic to stipulate a higher-level abstraction for data processing at the level of data processes illustrated by data-flow diagrams.

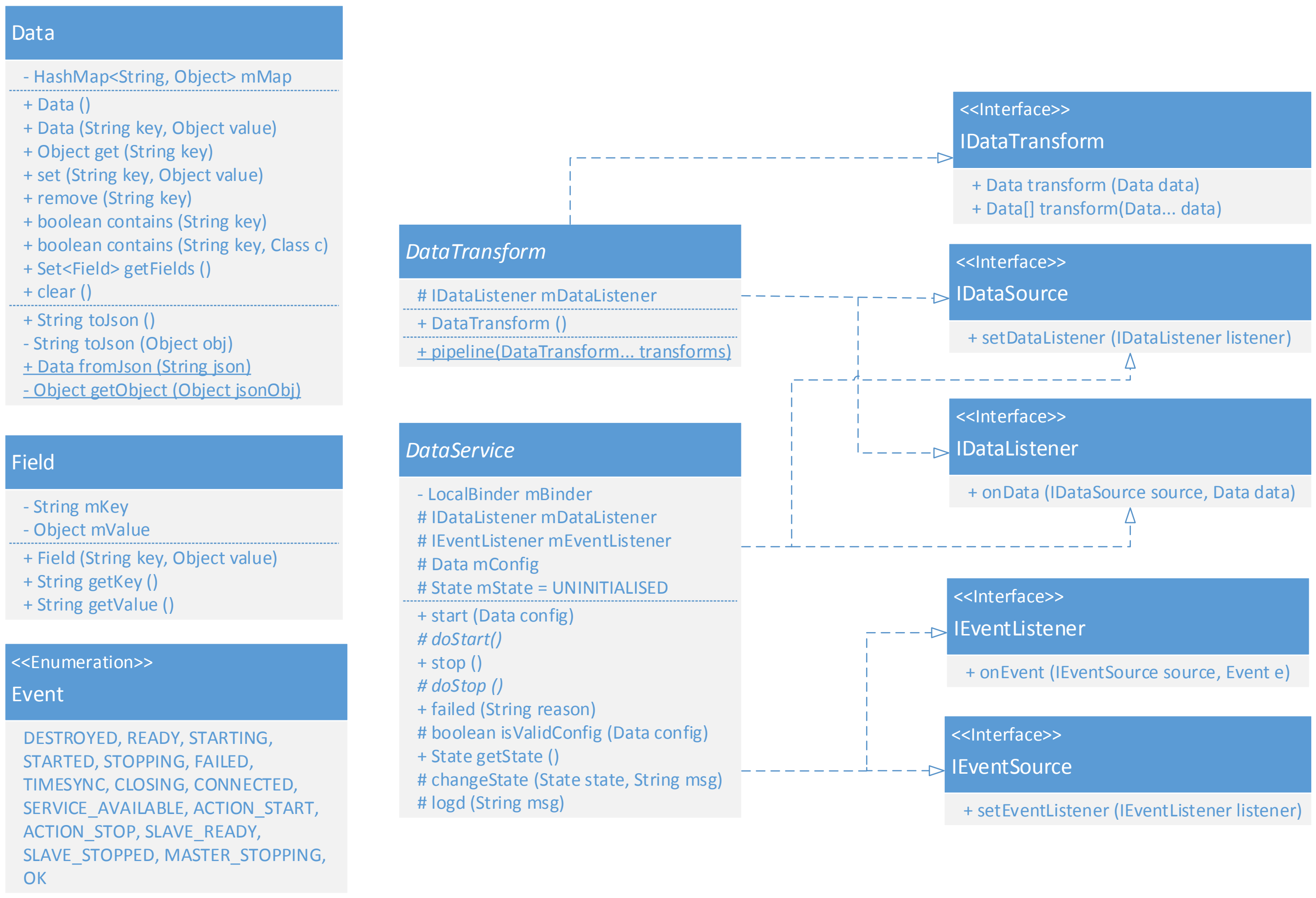

Data
- HashMap<String, Object> mMap
+ Data ()
+ Data (String key, Object value)
+ Object get (String key)
+ set (String key, Object value)
+ remove (String key)
+ boolean contains (String key)
+ boolean contains (String key, Class c)
+ Set<Field> getFields ()
+ clear ()
+ String toJson ()
- String toJson (Object obj)
+ Data fromJson (String json)
- Object getObject (Object jsonObj)
Field
- String mKey
- Object mValue
+ Field (String key, Object value)
+ String getKey ()
+ String getValue ()
<<Enumeration>>
Event
DESTROYED, READY, STARTING, STARTED, STOPPING, FAILED, TIMESYNC, CLOSING, CONNECTED, SERVICE_AVAILABLE, ACTION_START, ACTION_STOP, SLAVE_READY, SLAVE_STOPPED, MASTER_STOPPING, OK
DataTransform
# IDataListener mDataListener
+ DataTransform ()
+ pipeline(DataTransform... transforms)
DataService
- LocalBinder mBinder
# IDataListener mDataListener
# IEventListener mEventListener
# Data mConfig
# State mState = UNINITIALISED
+ start (Data config)
# doStart()
+ stop ()
# doStop ()
+ failed (String reason)
# boolean isValidConfig (Data config)
+ State getState ()
# changeState (State state, String msg)
# logd (String msg)
<<Interface>>
IDataTransform
+ Data transform (Data data)
+ Data[] transform(Data... data)
<<Interface>>
IDataSource
+ setDataListener (IDataListener listener)
<<Interface>>
IDataListener
+ onData (IDataSource source, Data data)
<<Interface>>
IEventListener
+ onEvent (IEventSource source, Event e)
<<Interface>>
IEventSource
+ setEventListener (IEventListener listener)

Figure 15: Classes of the Client Data-Processing Framework

At the core of the framework, the `Data` class is essentially a genericised implementation of a Java `HashMap`, exposing functionality for storing, retrieving and removing key-value pairs. The key is a `String` identifier, while the value can be data of any type. The `Data` class also provides methods for conversion to and from a JSON String representation of the object. This flexible implementation allows for representation and management of widely varied data using simple Java objects that are completely independent (each `Data` object completely encapsulates required type information) allowing for easy data processing and manipulation in Java, coupled with the ability to transform to a language-agnostic JSON representation.

```
...

Data data = new Data();         // Create empty data container
data.set("myNum", 1);           // Set a numeric value
data.set("myString", "Hello!"); // Set a string value
int myNum = data.get("myNum");  // Retrieve numeric value (Java Generics)

Data innerData = new Data();
data.set("anotherField", "World!"); // Define another data container and value

data.set("innerData", innerData); // Nested data

String json = data.toJson();    // Conversion to JSON representation

Data newData = Data.fromJson(json);  // Conversion from JSON representation

...
```

Just as the `Data` class is designed for flexible Java representation of data, the `IDataTransform` interface defines a basic frame for data manipulation. Semantically, this operation is defined as the input for the manipulation to be the parameter being passed in, and the returned `Data` object value is to be the transformed or manipulated result. The two signatures included allow for commonality between data-processing components but do not affect freedom of implementation.

Also partnered with the `Data` class are the `IDataListener` and `IDataSource` interfaces. These simplistic interfaces enable event-driven communication of `Data` objects between areas of data-processing code, while allowing those areas to retain complete loosely-coupled modularity.

These four components alone sufficiently cover the major aspects of data processing. `Data` objects provide stateless data representation, implementations of `IDataTransform` provide processes to manipulate Data objects, and implementation of `IDataSource` and `IDataListener` allows for modularised loosely-couple communication between data components. Despite this, the abstract classes `DataTransform` and `DataService` incorporate the previously discussed interfaces to provide structure for abstraction of data processing at a higher-level, and reduce boiler-plate implementation code when developing full data pipelines.

The abstract `DataTransform` class is the default essential realisation of the `IDataSource`, `IDataListener` and `IDataTransform` interfaces. An instance of this implementation operates as a base data-processing component, which responds to data (invocation of the `onData` method) by executing the transform method with the received data and then passing the transformed data to the registered IDataListener if one exists (invokes the onData method). This behaviour is illustrated by Figure 16.

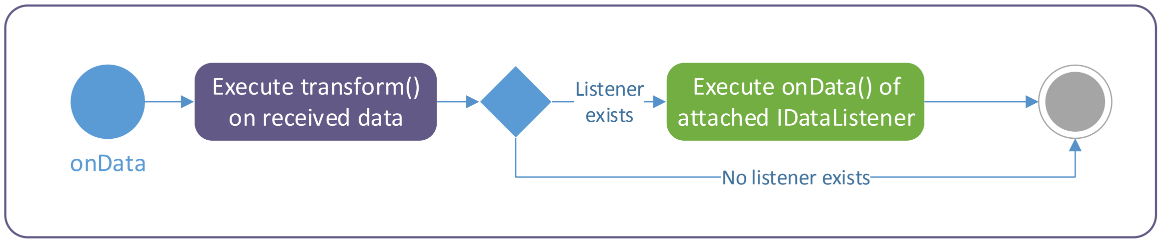

onData
Execute transform() on received data
Listener exists
Execute onData() of attached IDataListener
No listener exists

Figure 16: Activity Diagram of Data Transform

This behaviour allows for atomic data-processing components to be readily developed. Inheritance from the DataTransform class and implementation of the transform() method(s) with the desired data manipulation is all that is required to develop a stateless data process. As this behaviour is instance-based, parameterisation and state behaviour can be introduced using attributes. The following SetDataTransform code fragment illustrates the creation of a data-processing component by inheriting from the base DataTransform. The SetDataTransform is responsible for setting the value of a particular field to a particular value:

```java
public class SetDataTransform extends DataTransform
{
   private String mKey;
   private Object mValue;

   public SetDataTransform(String key, Object value)
   {
      mKey = key;
      mValue = value;
   }

   @Override
   public synchronized Data transform(Data data)
   {
      data.set(mKey, mValue);
      return data;
   }
}
```

The `DataService` class is designed to allow for the development of data pipeline components that accommodate long-running, persistent processes that may involve a lifecycle and may need to maintain connections to external sources or operating system services. By inheriting from the base Android `Service` class, `DataService` taps into the Android Service lifecycle for the ability to execute in the background. This class implements its own behaviours to handle the Android Service lifecycle and abstracts that lifecycle away from the framework. Another state-based lifecycle which is shown in Figure 17 is exposed as part of the framework and designed to more naturally reflect data processing.

This lifecycle revolves around the concept of providing the service with configuration data (represented using the `Data` class) that is utilised in parameterising initialisation. Once configuration and initialisation (connection to required resources - sensors, server, device etc.) is complete, the service should be ready to participate in the data pipeline by the event-driven movement of `Data` objects as semantically expected from `IDataSource` and `IDataListener` implementations. In the case of unexpected data or behaviour, the lifecycle is designed to handle movement into a failed state and then into a clean stop.

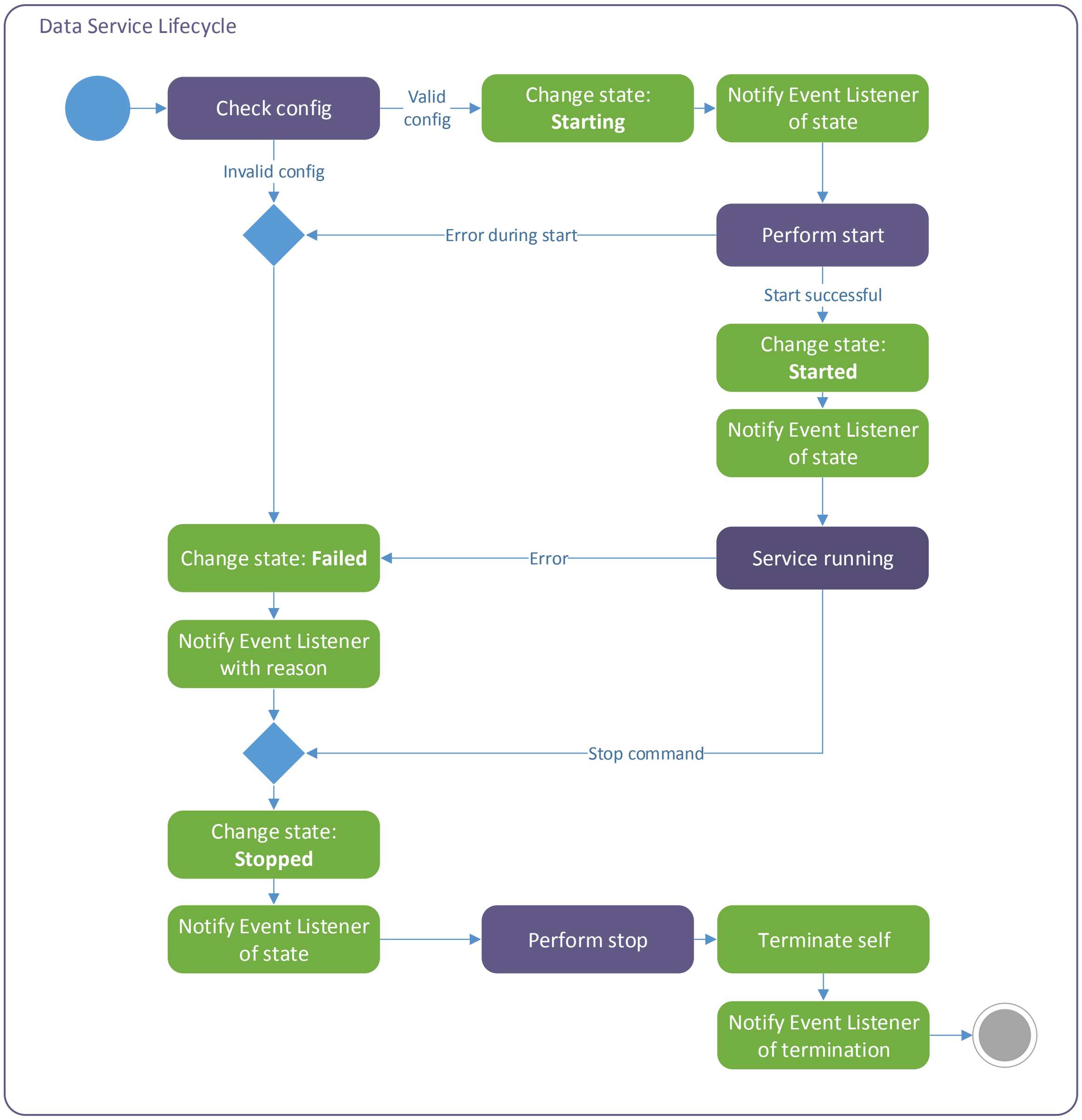


Figure 17: Activity Diagram of the Data Service Lifecycle

`DataService` classes communicate via the use of the Event enumeration alongside the `IEventSource` and `IEventListener` to allow for decoupled notification of state information and the passing of actions. Data-processing components are created using the `DataService` class in a similar manner as with `DataTransform`; inheriting from the `DataService` class and implementing the `isValidConfig()`, `doStart()`, `doStop()`, `onData()` and `onEvent()` methods with desired behaviour. The invocation of these methods in relation to the data service lifecycle is indicated in Figure 17.

6.6.1.3 Defined Data Components

With the previous section having discussed the foundational layers that frame the structure and operation of the data processing framework, the Defined Data Components layer introduces functional implementations for use in data-processing pipelines. Self-contained processing components such as data transforms and data services are defined using the high-level `DataTransform` and `DataService` partial implementations, while others are defined from a lower-level by realising the relevant framework interfaces. It is important to note that this layer is the lowest to involve concrete behaviours that are involved in final system processes, encompassing both a collection of generic, framework-defined processing components and any application-specific processing logic built upon the framework. As such, this layer forms a permeable boundary between the encapsulating framework and application layers of the client indicated in Figure 14.

The following discussion of the Defined Data Components layer exists to serve two purposes: to provide additional context regarding the framework layers by illustrating the use of the framework in defining data pipeline components; and to provide context for the application layer by cataloguing and explaining the functional data pipeline components defined and provided by the framework layer, which is discussed in paragraph 6.6.1.4.

The collection of data-processing components defined by the framework using the `DataTransform` abstraction are outlined in Table 2. Continuing the established ideology of self-contained, atomic operations, the data transforms provided by the framework account for a vast array of common data manipulation processes through singular operations and high parameterisability.

The purpose of these components is to streamline and shorten development time by promoting the modelling of data pipeline processes using modular components, as opposed to application-specific implementations of processing in code. Furthermore, the included implementations illustrate the intended use of the framework and can aid in the correct design and development of additional components.

While the focus of components developed using the `DataTransform` abstraction is on data manipulation and transformation, components built upon the `DataService` abstraction focus on interaction with external resources in order to integrate the use of them into the data pipeline. Below, in Table 3, the data services provided within the framework are outlined:

Transform	Purpose	Use in domain system
Set	Set a particular field in the data to a particular value	Addition of session ID metadata to data packets before caching
Remove	Removes a particular field from data	Removal of session ID metadata after caching to reduce size of HTTP packets
Deserialise	Extracts byte data from a defined field and deserialises that data into a `Data` object as result	Deserialisation of Data objects received over wireless connection from slave device
Field Copy	Copy value of defined field into another defined field	Preservation of timestamp data in aggregation steps
Field Modify	Prepend or append a given string onto a given field name	Identifying master and slave sensor data before aggregation
Field Rename	Removes a specified field and reassigns the value to another specified field name	Restoration of timestamp data after aggregation steps
Arithmetic	Performs the defined arithmetic operation on the value of a specified field in the data	Correction of slave timestamps with calculated time offset
Array Split	Retrieves an array value from the specified field and individually assigns the elements to given field names	Splitting of sensor data into individual axes (X, Y, Z)
Pack	Encapsulates all fields into a Data object value for the given field name	Preparation for aggregation of master and slave data
Unpack	Extracts a `Data` object value from a given field name and reassigns all fields into the overall data	Restoration of packed master and slave data following aggregation
Aggregator	Each field name in a given set that is present in the data is copied into internally held data. The internal data is returned and then reset when it is complete	Aggregating sensor data and master and slave data
Interval Aggregator	Extracts timestamp information from a given field name and aggregates based on the interval calculated	Aggregating sensor data and master and slave data
Array Collect	Adds received data into a collection. When collection reaches desired size the collection is returned as data and the collection reset	Collection of data into a data packet for reporting to remote server
Quaternion Difference	Extracts the information for two quaternions from given field names and calculates a difference quaternion which is stored back into the data by other given field names	Calculation of knee joint angle on completed data

Table 2: Outline of the data transform components included in the framework.

Service Name	Purpose
Sensor Sample	Samples data as provided by the Android sensor manager in accordance with the given configuration.
Bluetooth Connectivity	Enable use of the Android Bluetooth facilities to communicate data and events to another device operating the Bluetooth Connectivity Service.
Data Store	Enables caching of data pipeline results into persistence SQLite storage.
Remote Connectivity	Communicates with a remote RESTful API using HTTP requests for the transmission of data.

Table 3: Outline of the data service components included in the framework.

The operation of each of the above data services in Table 3 in relation to the base data service lifecycle will now be explored. These operations can be observed by identifying the implementations given for the data service lifecycle methods (that is the `isValidConfig()`, `doStart()`, `doStop()`, `onData()` and `onEvent()` methods).

6.6.1.3.1 Sensor Sampling Service

The sensor sampling service is designed to allow fully configurable sampling of sensor data, including sensor types (using Android definitions), field names for the aggregated data, and sampling rate. The behaviour of the sensor sampling service is illustrated in Figure 18. Upon initialisation, the sensor sampling service prepares the sampling process by communicating with the Android sensor manager in order to locate and access the required sensors. In the event of failure or expected stop, the service ensures data sampling is ceased.

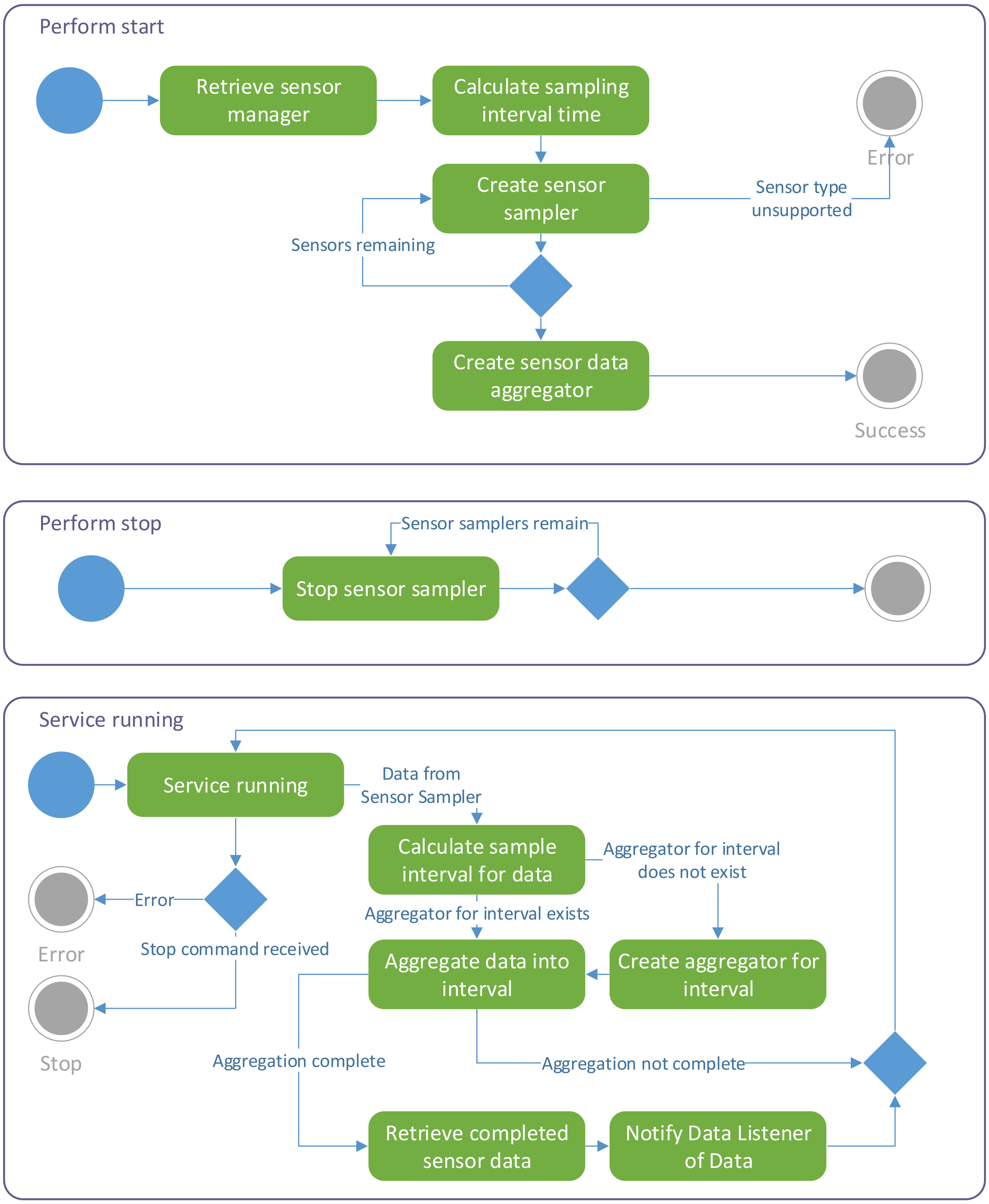


Figure 18: Sensor Sample Service Activity Diagram

The sensor sampling service is only designed for use at the beginning of a data pipeline, and thus has no facilities implemented to handle external incoming data. Utilising the `Interval-AggregatorDataTransform` introduced earlier the service amalgamates received sensor data and communicates completed data to the next stage of the pipeline using the event-driven behaviour required by the `IDataListener` interface.

6.6.1.3.2 Bluetooth Connectivity Service

In the same manner as most socket programming paradigms, the Bluetooth connectivity service does not operate in a symmetrical manner but rather involves a server-client process in which the server prepares a connection to which the client can locate and connect. The configuration for this service involves the role to be adopted (server or client) and in the case of the client the Bluetooth MAC address of the intended server is required. The behaviour of the Bluetooth connectivity service is illustrated in Figure 19 and Figure 20.

Following initial formation of the Bluetooth connection (completed in a dedicated thread), the connected devices undergo a time synchronisation step in order to quantify the time offset between them. This time offset is retained by the master device for future reference. At completion of the time synchronisation, the service has entered the 'started' state.

At the conclusion of execution, the service is responsible for closing the Bluetooth connection and terminating the related thread. As a courtesy the service attempts to notify the connected device of the impending closure.

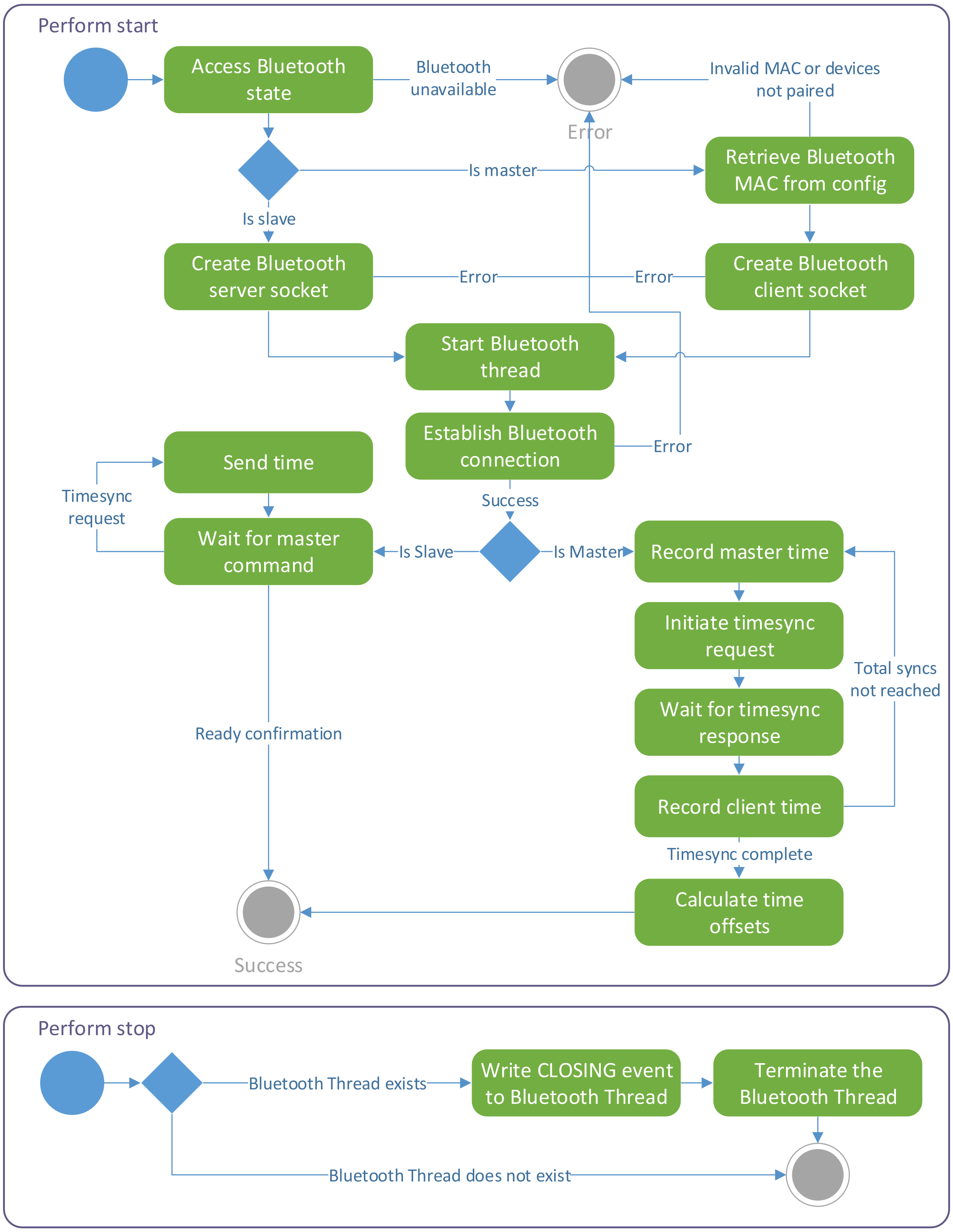


Figure 19: Bluetooth Connectivity Service Activity Diagram

While running, the service exists in a reactive state. Data and events received internally from the dedicated connection thread are deserialised, if necessary, and forwarded to the attached data component, while external data and events undergo serialisation and transportation through the Bluetooth connection. The service also responds to received events in accordance with the data service lifecycle, moving to the failed and stopped states when appropriate commands are received.

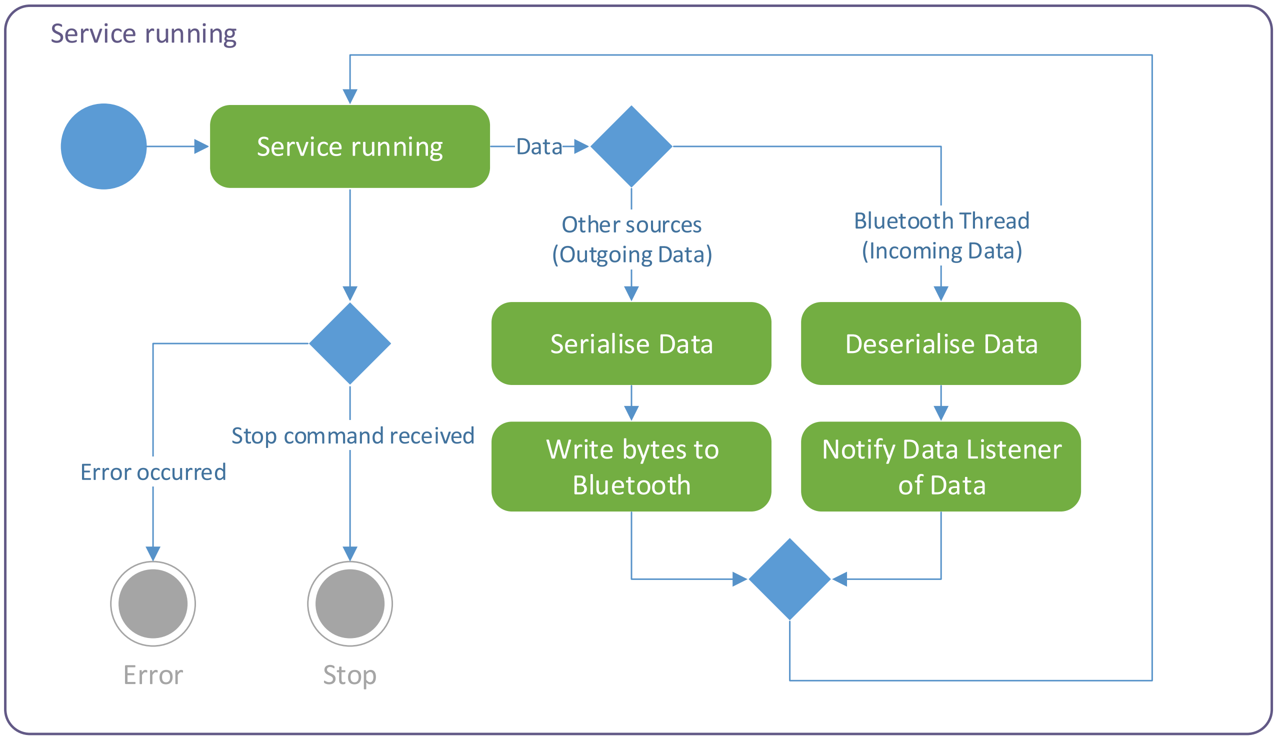


Figure 20: Bluetooth Connectivity Service Activity Diagram I

6.6.1.3.3 Remote Connectivity Service

The remote connectivity service is responsible for establishing that the URL indicated in the configuration exposes the correct RESTful API for receiving data. This is completed upon starting, by initiating a HTTP request to the URL and examining the response data. If the server responds correctly, the service starts correctly. The behaviour of the remote connectivity service is described in Figure 21

As the service does not hold any persistent connections or relevant states with the server, nothing is required upon stopping the service.

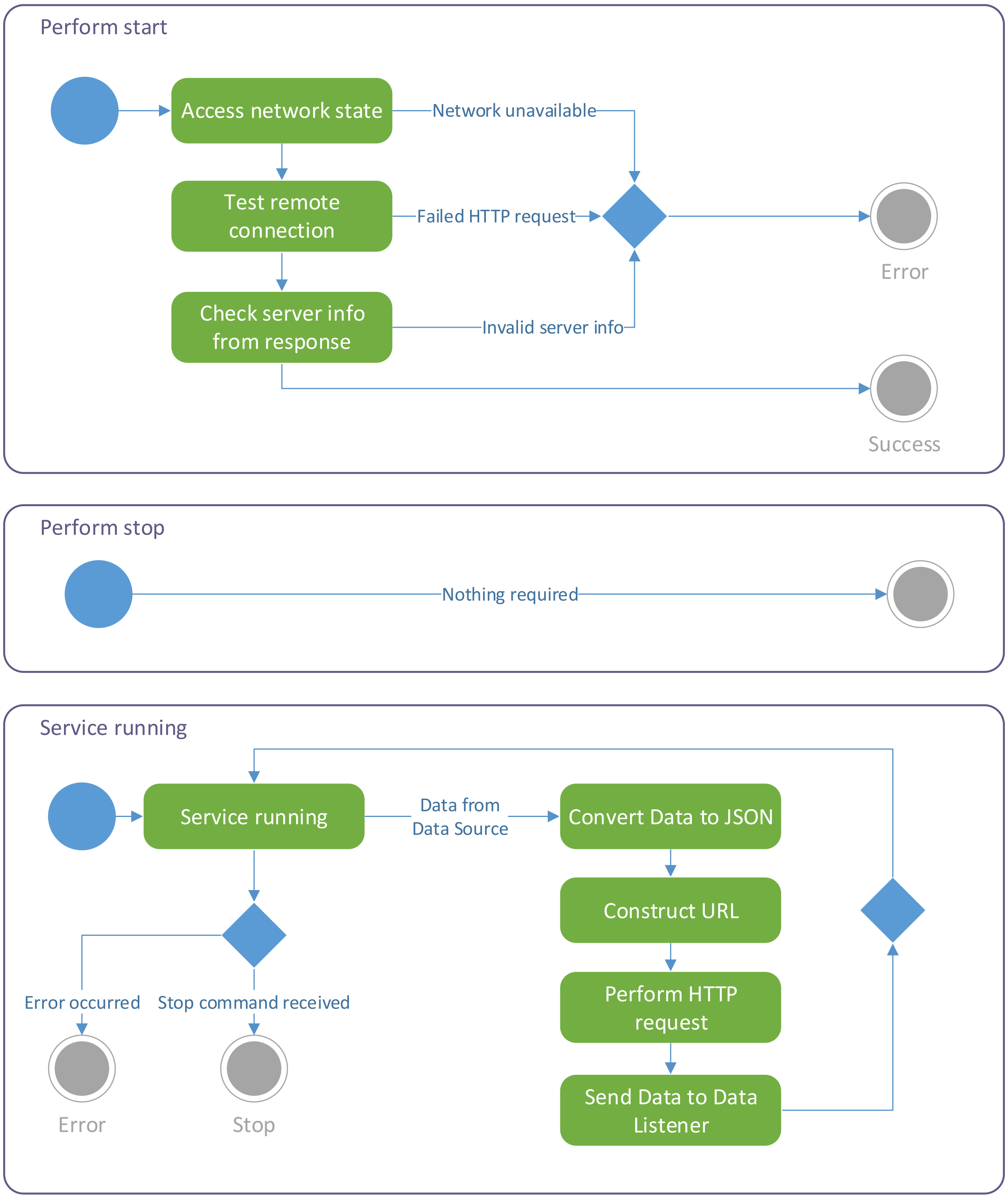


Figure 21: Remote Connectivity Service Activity Diagram II

While in a running state, the service serialises incoming data into a JSON representation and packages it into a HTTP submit request to the remote URL. The server response should return the JSON representation and an indication of success. On a success, the returned JSON is reconstituted into the Java representation and forwarded to the next pipeline component. It is important to note that HTTP requests are performed asynchronously and that this service does not forward data to the attached listener until the HTTP request has resolved.

6.6.1.3.4 Data Store Service

The data store service interacts with the Android libraries for SQLite databases in order to enable the ability for persistent caching of pipeline data. Upon initialisation and termination of the service, a connection to the relevant database is opened and closed respectively. The behaviour of the data store service is described in Figure 22

The service responds to received data by persisting it into the database and passing it to the attached pipeline component, without any modification. This silent reaction allows this service to be integrated into any area of the pipeline to seamlessly persist the results at that stage.

In addition to reactively responding to data, the service also exposes an API for managing the contents of the cache database through persistence, retrieval, and removal.

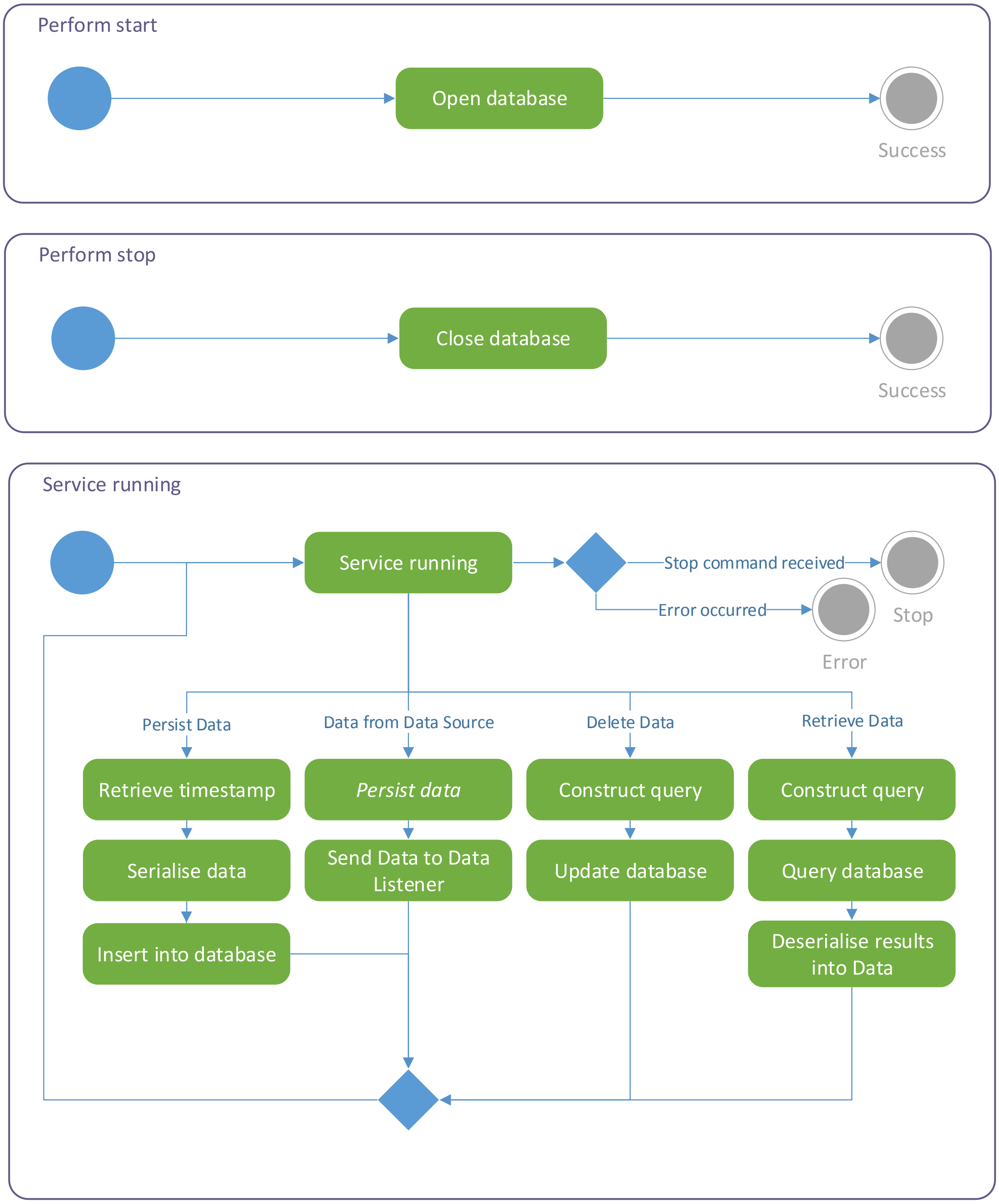


Figure 22: Data Store Service Activity Diagram

6.6.1.4 Coordinating Services

The next layer of the client architecture is responsible for coalescing data-processing components into a data pipeline. This layer comprises two implementations of the data service abstraction: the master service and the slave service. These services execute as part of the application on their respective devices and handle coordination and management of the data-processing components in that portion of the application's data pipeline, as defined by Figure 11.

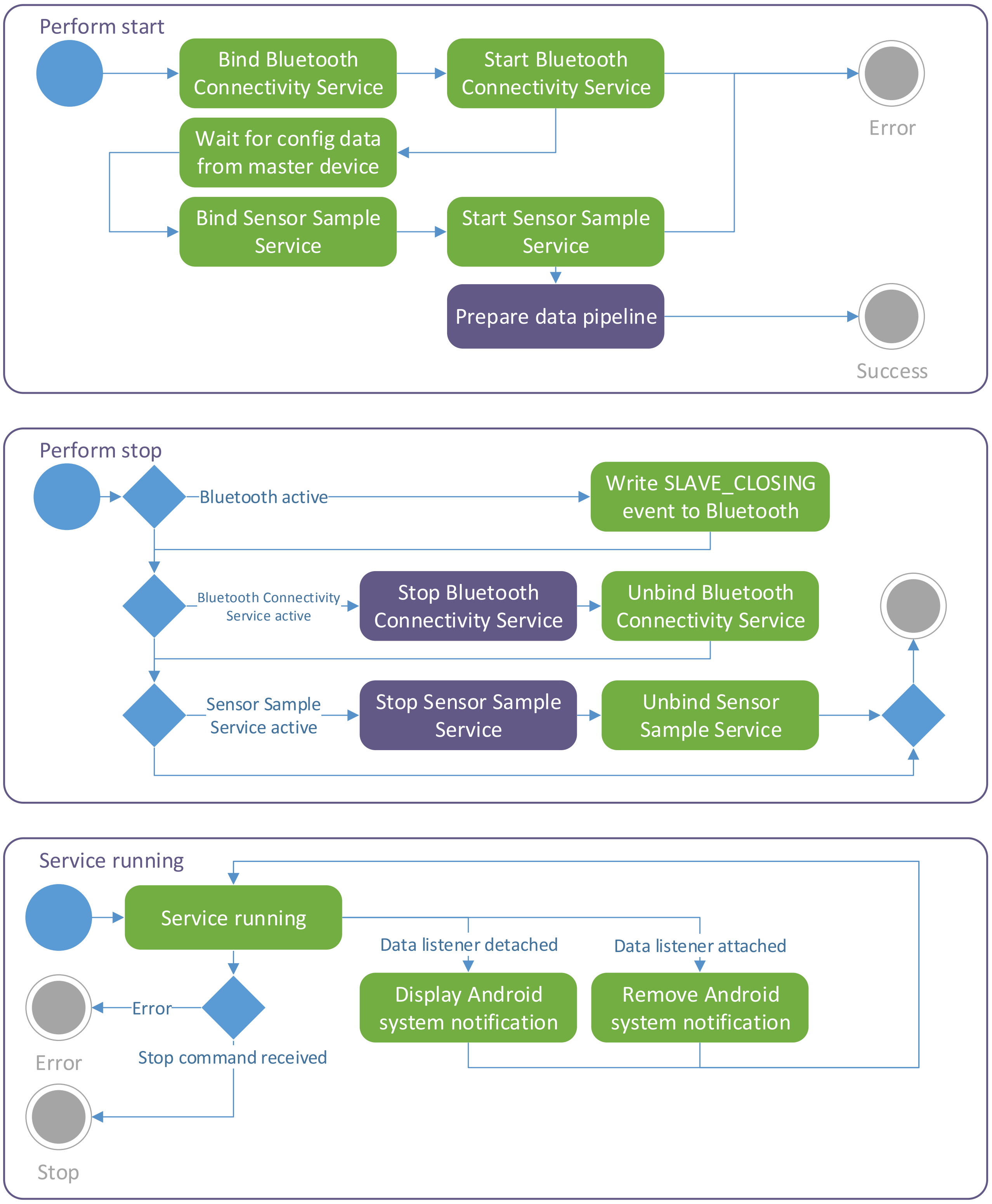
This layer is firmly within the overall application layer of the client, comprising of application-specific logic and behaviours. However, the coordinating services are still built upon the data service abstraction of the framework, and as such lifecycle management is still performed via the lifecycle methods, configuration data, and event actions. Furthermore, the coordinating services have no specific coupling to any kind of user interface as they retain the `IDataListener` perspective of an attached pipeline component.

6.6.1.4.1 Slave Service

Review of Figure 11 shows the simplistic nature of the slave device's operations. This is reflected by the behaviour of the slave service. The slave service neglects any given configuration data on start, instead adopting configuration data provided by the Bluetooth-connected master service for configuring the required sensor sample service. Upon correctly starting both the Bluetooth connectivity service and the sensor sample service, the slave service prepares its pipeline components before also entering the started state. The complete behaviours of the slave service are illustrated in Figure 23, and the structure of the slave pipeline is presented in Figure 26.

When termination of the slave is imminent, the slave service must coordinate the subsequent termination of the Bluetooth connectivity and sensor sample services that it is responsible for. Furthermore, it is important that the slave service first determines the current state of each service before attempting termination, in order to avoid feedback loops due to event notifications of state, as it may have been the failure (unexpected termination) of a data service that prompts the termination of the coordinating slave service.

While in a running state, the coordinating slave service has no immediate concerns apart from responding to relevant events. As a concession to the Android system, the service monitors the state of the data listener and enters and exits foreground service state in the Android system as appropriate.



Perform start
Bind Bluetooth Connectivity Service
Start Bluetooth Connectivity Service
Error
Wait for config data from master device
Bind Sensor Sample Service
Start Sensor Sample Service
Prepare data pipeline
Success
Perform stop
Bluetooth active
Write SLAVE_CLOSING event to Bluetooth
Bluetooth Connectivity Service active
Stop Bluetooth Connectivity Service
Unbind Bluetooth Connectivity Service
Sensor Sample Service active
Stop Sensor Sample Service
Unbind Sensor Sample Service
Service running
Service running
Data listener detached
Data listener attached
Display Android system notification
Remove Android system notification
Error
Error
Stop command received
Stop

Figure 23: Slave Service Activity Diagram

6.6.1.4.2 Master Service

Conversely, review of Figure 11 reveals the complex nature of the master device's operations. The master service is responsible for coordinating an extensive portion of the system's data flow and also managing the extensive configurability of the components involved. Unlike the slave service, the master service expects a completed configuration upon start, and uses this during an extensive initialisation. As the starting and binding of Android services is not a synchronous operation, the master service binds all required data services and awaits the relevant callbacks. Upon successful start of all involved services, the configuration is forwarded to the slave service via the Bluetooth connectivity service and the pipeline components are prepared. The behaviours of the master service are defined in Figure 24 and Figure 25, with the contents of the master pipeline being illustrated by Figure 26.

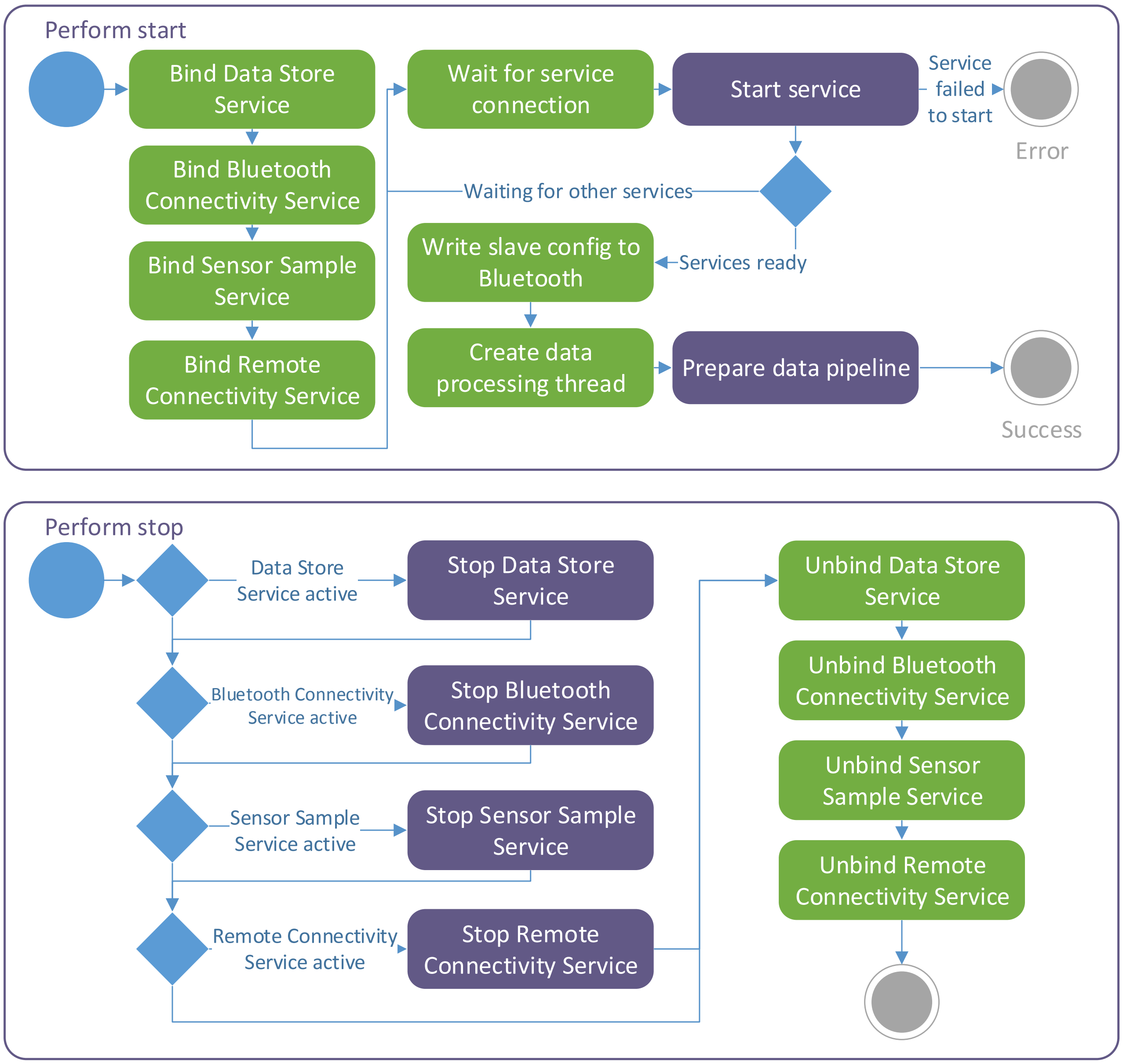
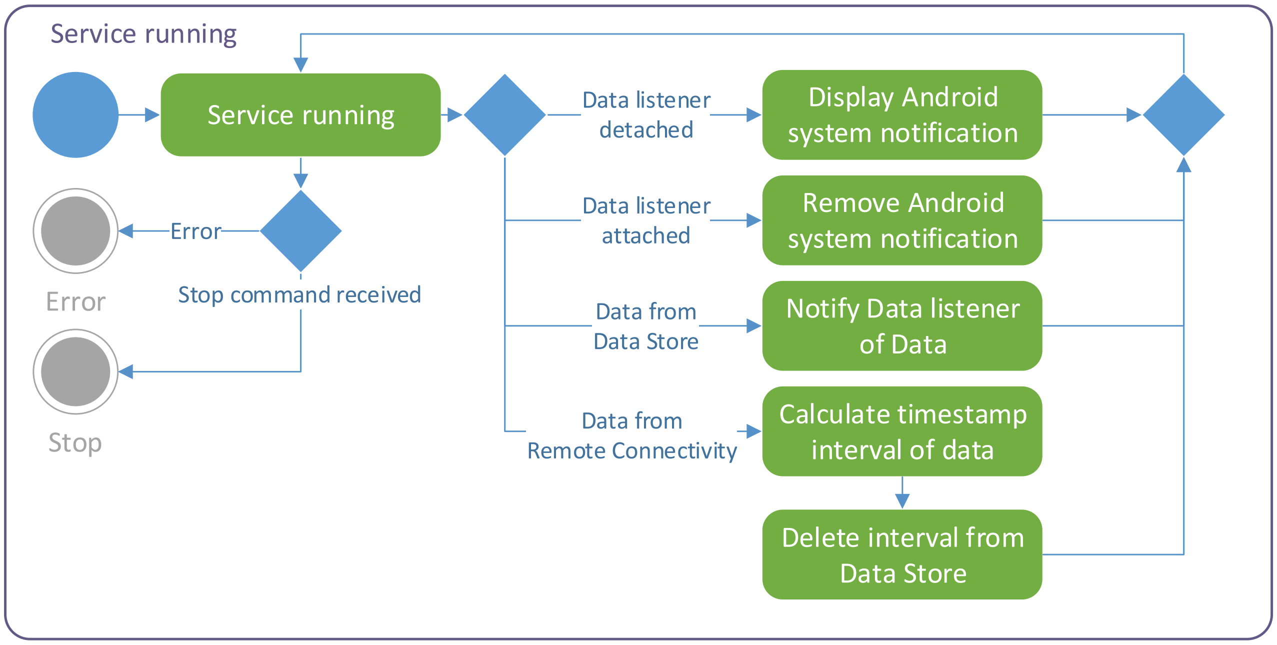

Perform start
Bind Data Store Service
Bind Bluetooth Connectivity Service
Bind Sensor Sample Service
Bind Remote Connectivity Service
Wait for service connection
Start service
Service failed to start
Error
Waiting for other services
Services ready
Write slave config to Bluetooth
Create data processing thread
Prepare data pipeline
Success
Perform stop
Data Store Service active
Stop Data Store Service
Bluetooth Connectivity Service active
Stop Bluetooth Connectivity Service
Sensor Sample Service active
Stop Sensor Sample Service
Remote Connectivity Service active
Stop Remote Connectivity Service
Unbind Data Store Service
Unbind Bluetooth Connectivity Service
Unbind Sensor Sample Service
Unbind Remote Connectivity Service

Figure 24: Master Service Activity Diagram - Starting and Stopping

In the same manner as the slave service, the master service is responsible for coordinating the termination of multiple data services during its own termination. The master service performs this task in the same manner, ensuring activity of the service before attempting termination in order to avoid any event-based feedback loops.

Once the master data pipeline is initialised and operational, the master service only has a small involvement with the resulting data. Data is transported to the master service via two routes (see Figure 11), with individual data points proceeding from the cache persistence being passed to the attached listener (the user interface), while data forwarded from the remote connectivity service is treated as confirmation of receipt by the server and is used to retroactively clear the cache. Similarly to the slave service, the master service responds to events and manages its foreground Android service state.



Service running
Service running
Error
Error
Stop command received
Stop
Data listener detached
Display Android system notification
Data listener attached
Remove Android system notification
Data from Data Store
Notify Data listener of Data
Data from Remote Connectivity
Calculate timestamp interval of data
Delete interval from Data Store

Figure 25: Master Service Activity Diagram - Running

While both coordinating services are initialised and active, data is processed through the system by way of the chain of data-processing components prepared by each coordinating service. The complete flow of data processing steps is shown in Figure 26. When this flow is compared with the system data flow (Figure 11) a strong resemblance emerges, illustrating how the data processing framework of the system simplifies and streamlines the development of data-processing applications by promoting a pipeline-centric view of the data operations, transformations, and transportation.

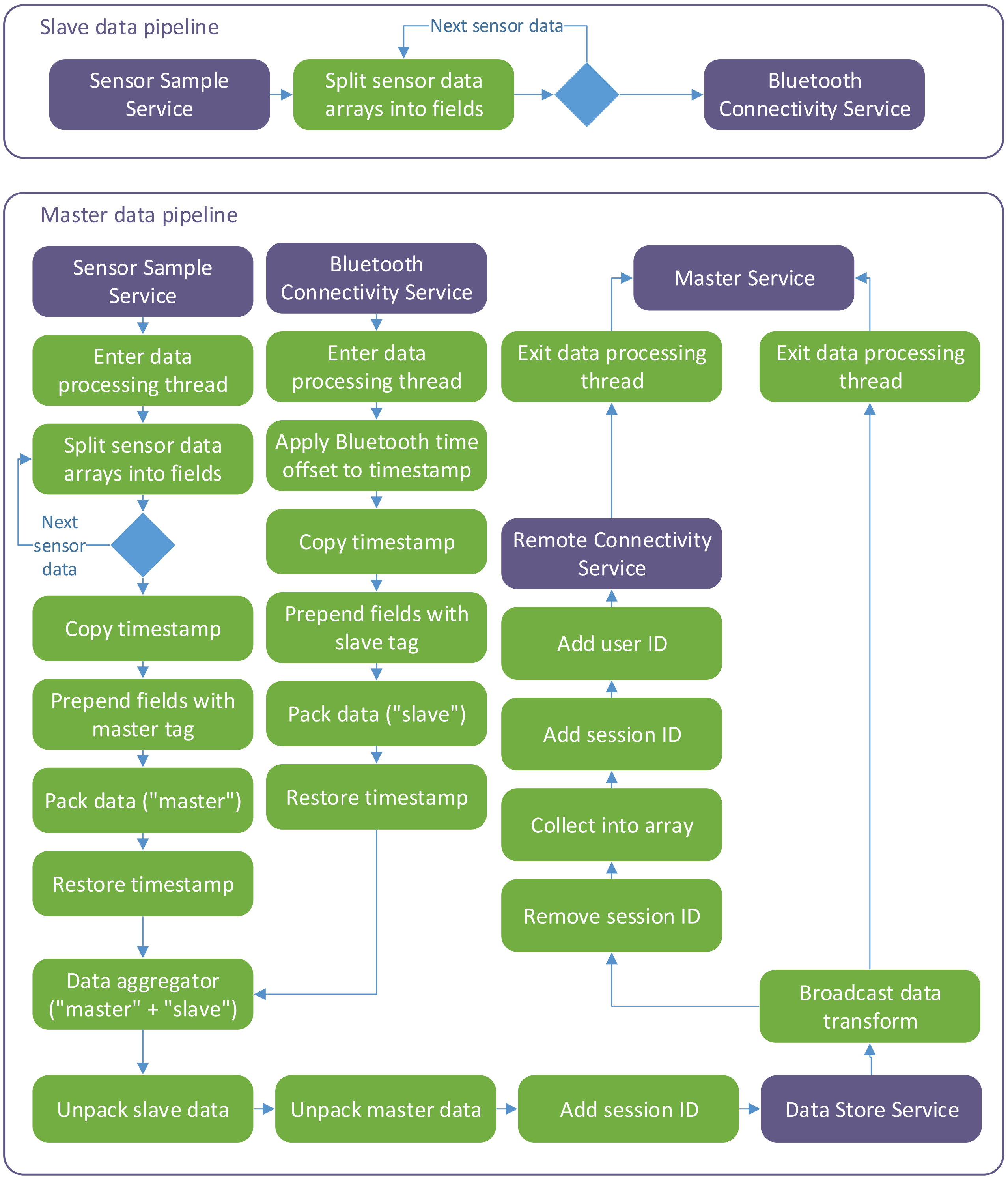

Slave data pipeline
Sensor Sample Service
Split sensor data arrays into fields
Next sensor data
Bluetooth Connectivity Service
Master data pipeline
Sensor Sample Service
Enter data processing thread
Split sensor data arrays into fields
Next sensor data
Copy timestamp
Prepend fields with master tag
Pack data ("master")
Restore timestamp
Data aggregator ("master" + "slave")
Bluetooth Connectivity Service
Enter data processing thread
Apply Bluetooth time offset to timestamp
Copy timestamp
Prepend fields with slave tag
Pack data ("slave")
Restore timestamp
Master Service
Exit data processing thread
Remote Connectivity Service
Add user ID
Add session ID
Collect into array
Remove session ID
Exit data processing thread
Broadcast data transform
Unpack slave data
Unpack master data
Add session ID
Data Store Service

Figure 26: Data Pipelines of the Slave and Master Services

6.6.1.5 User Interface

The user interface of the client sensor network has two responsibilities:

- Present Android activities for user interaction
- Allow for configuration of data sampling and application behaviour (master only)
- Prepare correct configuration data from Android preferences (master only)
- Control and monitor the lifecycle of the attached coordinating service
- Respond to data received from the attached coordinating service - knee angle warnings

The user interfaces are defined using the Android libraries for UI activities, and fall firmly into the application-specific layer of the system. However, in order to unobtrusively interact with the coordinating services for sending and receiving data and events, the Android activities implement many of the relevant interfaces provided by the framework layer. This approach avoids any tight-coupling for the coordinating services, allowing them to remain independent of the user interface implementation.

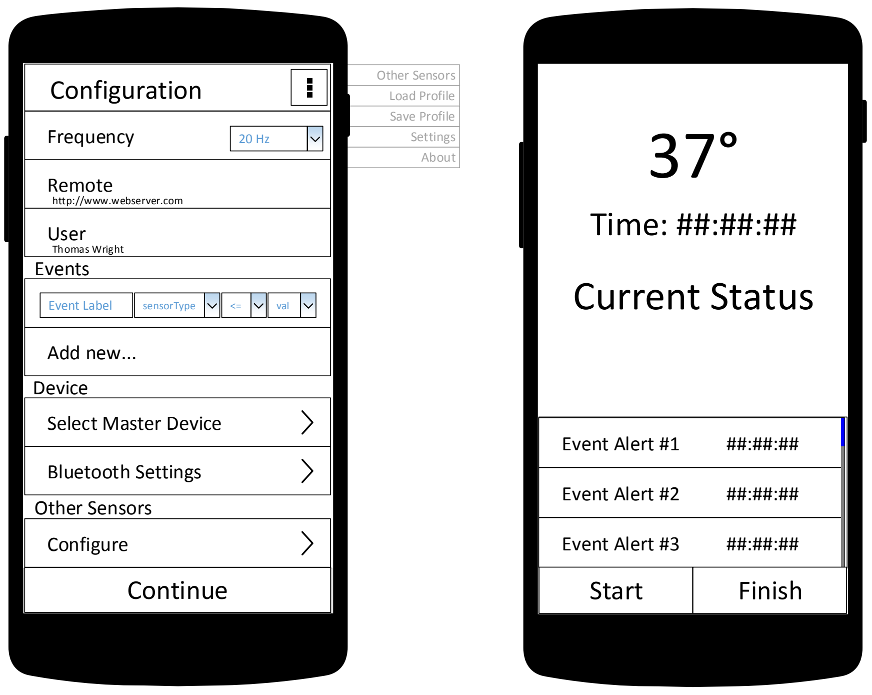


Figure 27: Client Master User Interface

The master interface provides an Android preference screen for simple, persistent configuration of application settings. These Android preferences are compiled into a configuration data object for initiation of the attached master service. Further management of the master service lifecycle is done by passing an event rather than the use of a method call, to further limit the coupling between the coordination service and interface layers.

The master interface attaches itself as the data listener for the coordinating master service, making it the last data component in the pipeline. The interface analyses the resultant data and handles generation of knee angle warnings as per the component specification (subsection 5.2)

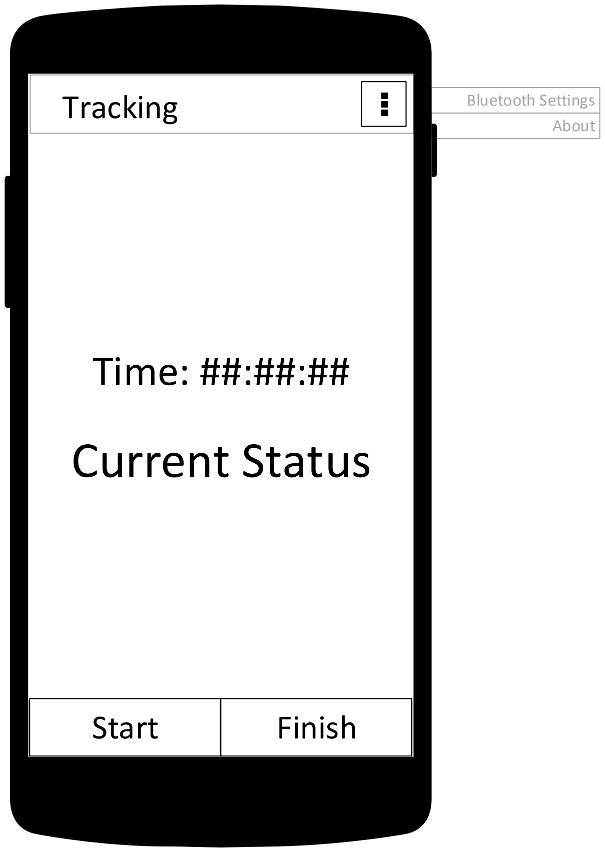


Figure 28: Client Slave User Interface

The slave interface is even more simplistic, with only controls to initiate and terminate the underlying coordinating slave service. Configuration data is not the responsibility of the slave service due to the coordinating slave service receiving sensor configuration data directly from the master service on the paired device. The slave interface is solely responsible for allowing user interaction to begin and end the lifecycle of the slave service.

6.6.1.6 Quaternion Class Library

The Quaternion Class library stands apart from the main client architecture, but is critical in supporting calculations required by the data-processing pipeline of the system.

The Quaternion Class library is implemented following the discussion of section 4. The class library provides many typical vector operations as well as several of the specialised Quaternion operations outlined there. To ensure numeric stability and ease-of-use, the class is designed such that these operations are provided as member functions of class instantiations, where each member function (e.g. Hamilton Product) takes as parameter the RHS (right-hand-side) operand, if any, and yields a new class instance representing the result of the operation.

The Client Sensor Network uses the Quaternion Class Library to create a difference Quaternion representing the rotational distance between two fixed orientations. The magnitude of the difference Quaternion is then taken to determine the current angle subtended by each orientation vector. The relevant methods are shown in brief below:

```
// Constructor
public Quaternion(float[] rv)
{
    this.q = new float[4];
    getQuaternionFromVector(rv);
}

private void getQuaternionFromVector(float[] rv)
{
    // Take the vector w component if it exists
    if (rv.length == 4) {
        q[0] = rv[3];
    }
    // Calculate the w component as sqrt(1 - |rv|)
    else {
        q[0] = 1 - rv[0]*rv[0] - rv[1]*rv[1] - rv[2]*rv[2];
        q[0] = (q[0] > 0) ? (float)Math.sqrt(q[0]) : 0;
    }
    // Set the vector component
    q[1] = rv[0];
    q[2] = rv[1];
    q[3] = rv[2];
}

// Calculate a difference Quaternion between this and qPrev
public Quaternion Difference(Quaternion qPrev)
{
    return this.Conjugate().HamiltonProduct(qPrev);
}
```

```
// Get the conjugate of this Quaternion
public Quaternion Conjugate()
{
    return new Quaternion(q[0], -q[1], -q[2], -q[3]);
}

// Calculate the Hamilton Product of this and another Quaternion q2
public Quaternion HamiltonProduct(Quaternion q2)
{
    return new Quaternion(
        -q[1] * q2.X() - q[2] * q2.Y() - q[3] * q2.Z() + q[0] * q2.W(),
         q[1] * q2.W() + q[2] * q2.Z() - q[3] * q2.Y() + q[0] * q2.X(),
        -q[1] * q2.Z() + q[2] * q2.W() + q[3] * q2.X() + q[0] * q2.Y(),
         q[1] * q2.Y() - q[2] * q2.X() + q[3] * q2.W() + q[0] * q2.Z());
}
```

For efficiency, the class library provides a subset of the functionality as streamlined static methods that take as parameter float arrays representing the orientation vectors, and perform the operations "in place" without creating new class instantiations. These are given:

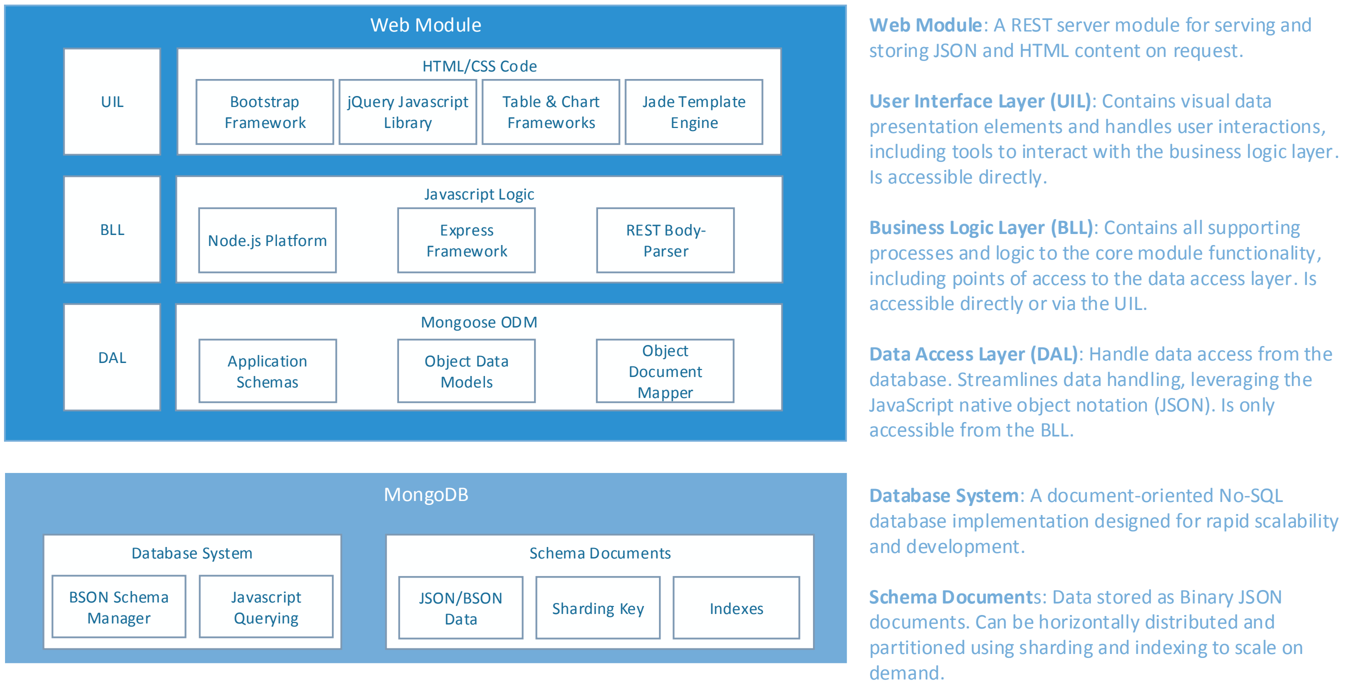
```
public static void getDifferenceQuaternion(float[] res, float[] vNew, float[] vPrev)
{
    if (vNew.length == 4)
    {
        vNew[1] = -vNew[1];
        vNew[2] = -vNew[2];
        vNew[3] = -vNew[3];

        res[0] = -vNew[1] * vPrev[1] - vNew[2] * vPrev[2] - vNew[3] * vPrev[3]
                 + vNew[0] * vPrev[0];
        res[1] =  vNew[1] * vPrev[0] + vNew[2] * vPrev[3] - vNew[3] * vPrev[2]
                 + vNew[0] * vPrev[1];
        res[2] = -vNew[1] * vPrev[3] + vNew[2] * vPrev[0] + vNew[3] * vPrev[1]
                 + vNew[0] * vPrev[2];
        res[3] =  vNew[1] * vPrev[2] - vNew[2] * vPrev[1] + vNew[3] * vPrev[0]
                 + vNew[0] * vPrev[3];
    }
    else
    {
        float[] qNew = new float[4];
        float[] qPrev = new float[4];

        getQuaternionFromVector(qNew, vNew);
        getQuaternionFromVector(qPrev, vPrev);

        qNew[1] = -qNew[1];
        qNew[2] = -qNew[2];
        qNew[3] = -qNew[3];

        res[0] = -vNew[1] * vPrev[1] - vNew[2] * vPrev[2] - vNew[3] * vPrev[3]
                 + vNew[0] * vPrev[0];
        res[1] =  vNew[1] * vPrev[0] + vNew[2] * vPrev[3] - vNew[3] * vPrev[2]
                 + vNew[0] * vPrev[1];
        res[2] = -vNew[1] * vPrev[3] + vNew[2] * vPrev[0] + vNew[3] * vPrev[1]
                 + vNew[0] * vPrev[2];
        res[3] =  vNew[1] * vPrev[2] - vNew[2] * vPrev[1] + vNew[3] * vPrev[0]
                 + vNew[0] * vPrev[3];
    }
}

public static float getQuaternionMagnitude(float[] q)
{
    float lenSquared = q[1] * q[1] + q[2] * q[2] + q[3] * q[3];
    return (lenSquared > 0 ? 2 * (float)Math.atan2(Math.sqrt(lenSquared), q[0]) :
                             (float) Math.atan2(0, q[0]));
}

public static void getQuaternionFromVector(float[] q, float[] rv)
{
    // Take the vector w component if it exists
    if (rv.length == 4) {
        q[0] = rv[3];
    }
    // Calculate the w component as sqrt(1 - |rv|)
    else {
        q[0] = 1 - rv[0]*rv[0] - rv[1]*rv[1] - rv[2]*rv[2];
        q[0] = (q[0] > 0) ? (float)Math.sqrt(q[0]) : 0;
    }
    // Set the vector component
    q[1] = rv[0];
    q[2] = rv[1];
    q[3] = rv[2];
}
```

6.6.2 Web Application Server

The Web Application Server is implemented using MongoDB, a document-oriented NoSQL database management system, to handle the persisted data, with Express.js, a flexible web application development framework built on top of the Node.js server runtime environment, acting as the primary middleware for handling HTTP requests. Serving as the go-between for Express.js and MongoDB is the Mongoose ODM (Object Document Mapper) plugin.

6.6.2.1 Architecture

The Web Server Application defines a number of routes (URLs with attributes) that are each registered to a handler (a function that is called any time a web client sends a HTTP request to the route) and serve as the only points of access to the application. The application offers public routes in two forms: the BLL (Business Logic Layer), and the UIL (User Interface Layer). The routes in the BLL define points of access to the underlying persisted data, such as storage and retrieval requests, while the routes in the UIL form a secondary layer on top of the BLL, providing presentational HTML views of the data accessible via the BLL routes. The UIL and BLL therefore serve as the only points of access to the DAL (Data Access Layer), allowing the DAL to be obfuscated such that direct access to the persisted data is impossible. This allows for a more standardised API, as well as greater security. This component architecture is shown in greater detail in Figure 29.



Web Module: A REST server module for serving and storing JSON and HTML content on request.

User Interface Layer (UIL): Contains visual data presentation elements and handles user interactions, including tools to interact with the business logic layer. Is accessible directly.

Business Logic Layer (BLL): Contains all supporting processes and logic to the core module functionality, including points of access to the data access layer. Is accessible directly or via the UIL.

Data Access Layer (DAL): Handle data access from the database. Streamlines data handling, leveraging the JavaScript native object notation (JSON). Is only accessible from the BLL.

Database System: A document-oriented No-SQL database implementation designed for rapid scalability and development.

Schema Documents: Data stored as Binary JSON documents. Can be horizontally distributed and partitioned using sharding and indexing to scale on demand.

Figure 29: Web Application Server Architecture Diagram

6.6.2.2 Design

The BLL defines a RESTful API built around the CRUD (Create, Read, Update, Delete) methodology for interacting with the application's persisted resource data. As such, a unique route is provided to access each resource for each of the applicable CRUD operations. As the data model given in subsection 6.5 provides the notion of a session that is appended to in "chunks", the CRUD specification is therefore given by Table 4; here, a collection represents a list of resources, an aggregated resource indicates a number of combined resources (as in the case of a session document composed of its parts), and an atomic resource indicates a single document.

Resource/Type	GET	PUT	POST	DELETE
Collection /api/session/	List all the sessions in the collection and their metadata	N/A	N/A	N/A
Aggregated Resource /api/session/:id	Aggregate all the parts belonging to the session given by :id and provide the complete data	N/A	Insert in order a single document part to the session given by :id, or create the session if it does not exist.	Delete all documents in the session given by :id.
Atomic Resource /api/session/ :id/:part	Get the single document specified by :part of a session specified by :id	N/A	N/A	N/A
Collection /api/user/:id	List all the data belonging to the user specified by :id–in this case, a list of sessions	N/A	N/A	N/A

Table 4: Web Application Server CRUD Route Specification

The BLL additionally offers another route for the purpose of identifying the server to connecting sensor clients. This method simply lists properties of the server, including a flag that indicates that the server application is compliant with the forms of requests listed above. The function of each of these routes is shown in greater detail by the activity diagrams in Figure 30 and Figure 31.

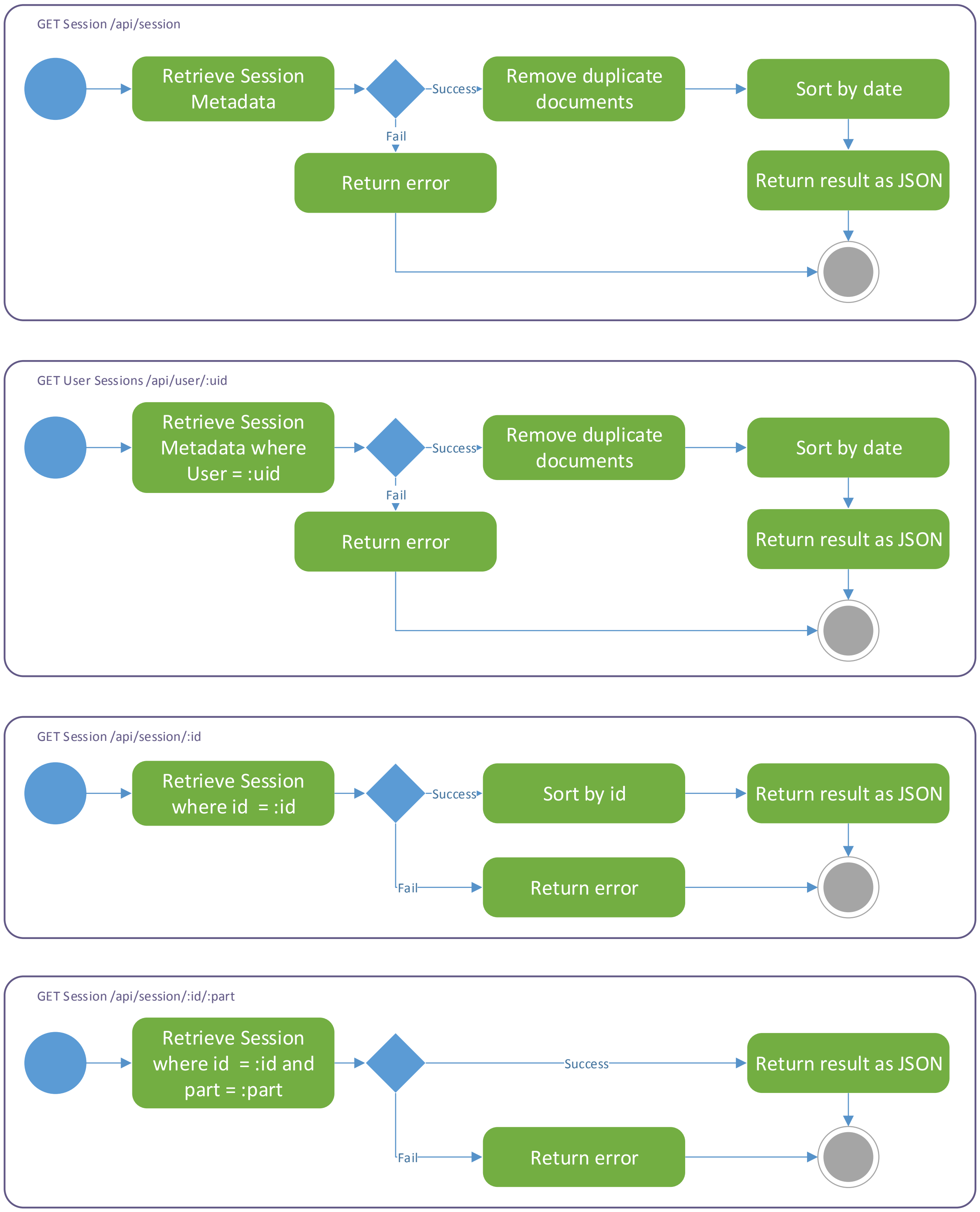

GET Session /api/session
Retrieve Session Metadata
Success
Fail
Remove duplicate documents
Sort by date
Return error
Return result as JSON
GET User Sessions /api/user/:uid
Retrieve Session Metadata where User = :uid
Success
Fail
Remove duplicate documents
Sort by date
Return error
Return result as JSON
GET Session /api/session/:id
Retrieve Session where id = :id
Success
Sort by id
Return result as JSON
Fail
Return error
GET Session /api/session/:id/:part
Retrieve Session where id = :id and part = :part
Success
Return result as JSON
Fail
Return error

Figure 30: Web Application Server API Routes Activity Diagram I

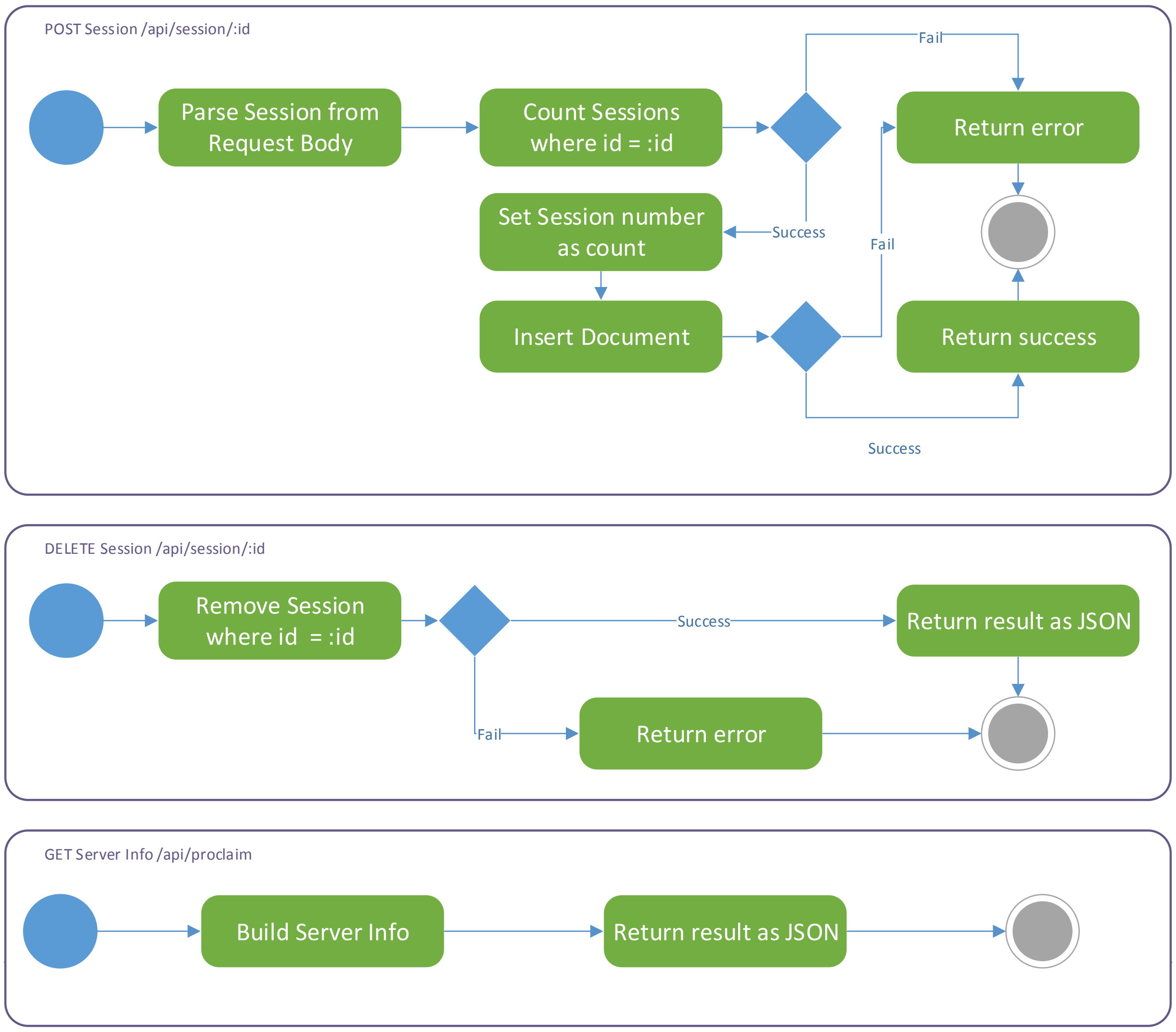

POST Session /api/session/:id
Parse Session from Request Body
Count Sessions where id = :id
Fail
Return error
Set Session number as count
Success
Fail
Insert Document
Return success
Success
DELETE Session /api/session/:id
Remove Session where id = :id
Success
Return result as JSON
Fail
Return error
GET Server Info /api/proclaim
Build Server Info
Return result as JSON

Figure 31: Web Application Server API Routes Activity Diagram II

The UIL is built on top of the BLL and is designed to provide visual controls to execute the BLL functionality, as well as offer presentational views of the data returned. The UIL uses the Jade template engine to generate HTML pages, using CSS for page styling. The layer provides routes for viewing all stored sessions, and for specific information regarding a session. The high level process of the complete GUI offered by the UIL is shown in Figure 32.

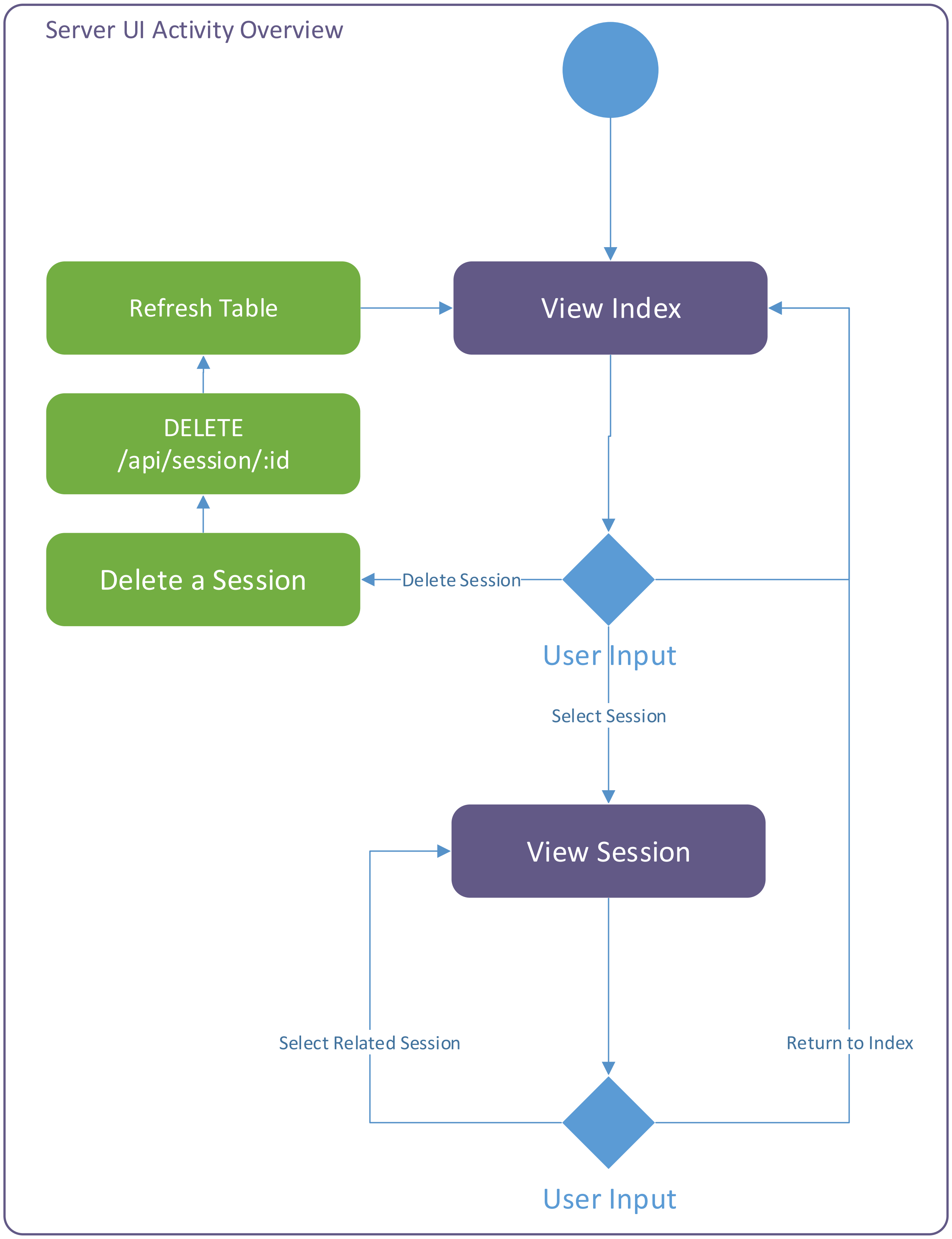


Figure 32: Web Application Server High-Level GUI Activity Diagram

From the Index route, a web client may view all of the sessions as well as the time they were commenced and the user the session pertains to–loading this data via the relevant route in the BLL. Actions are also provided to view a selected session in greater detail (which redirects to the session route), or to delete the session (leveraging the relevant route in the BLL). The sequence of processes in returning the rendered HTML page when the route is accessed is shown in Figure 33.

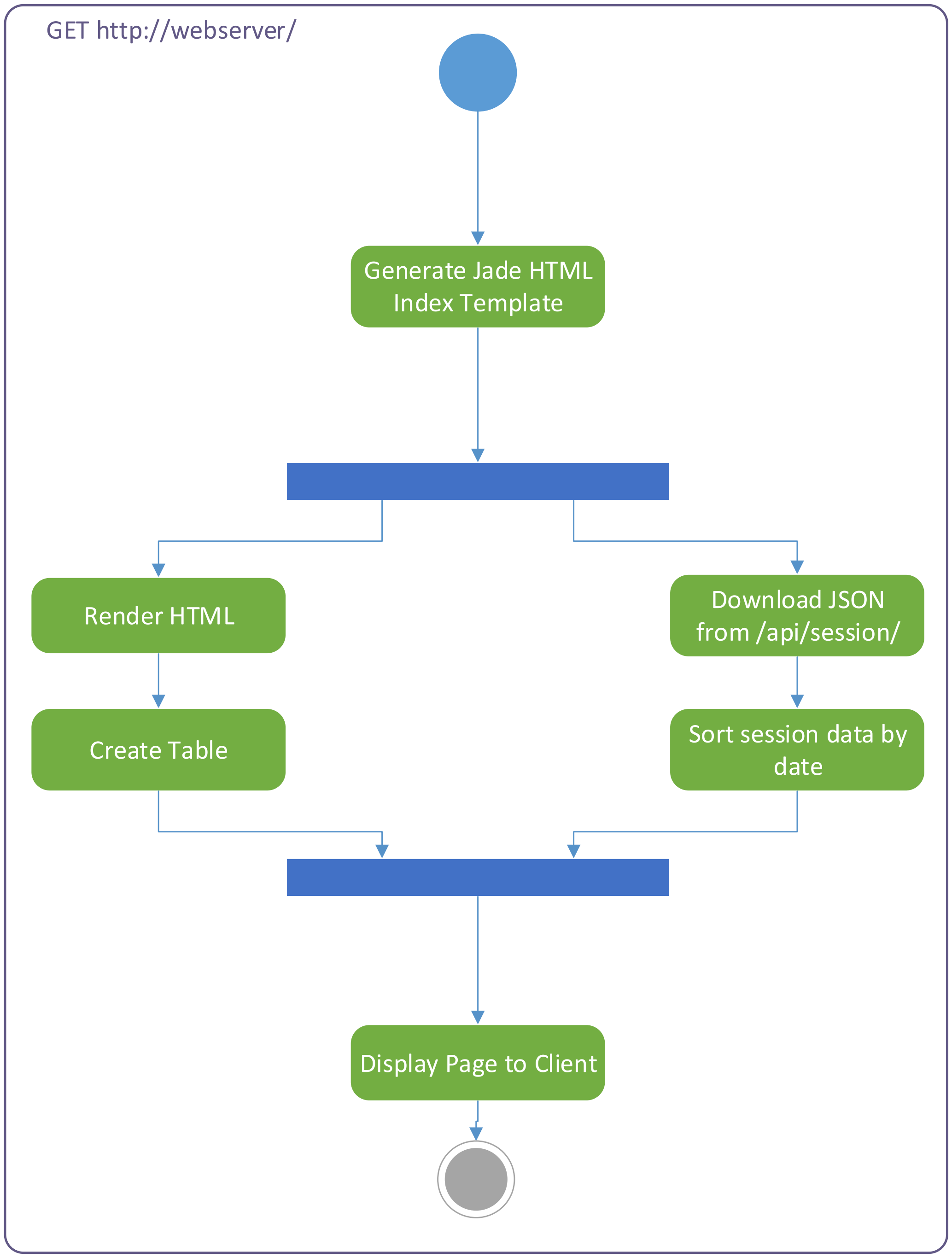
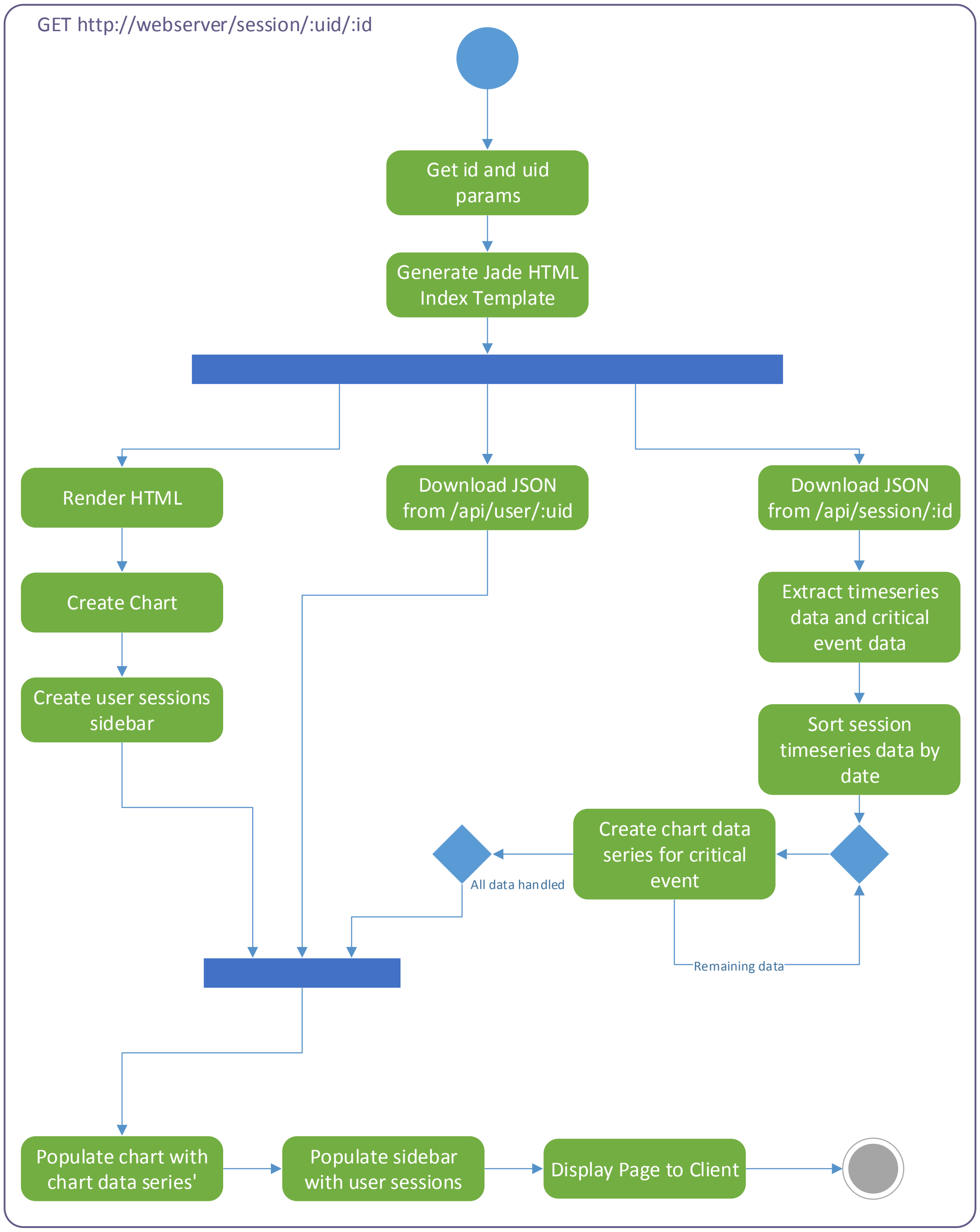

GET http://webserver/
Generate Jade HTML Index Template
Render HTML
Download JSON from /api/session/
Create Table
Sort session data by date
Display Page to Client

Figure 33: Web Application Server Index GUI Activity Diagram

In the Session route, displayed when a web client is viewing a session in full, the client can view each series of sensor data stored in the session, in order to provide quick analysis of the data. The client may also view related sessions, including other sessions previously logged by the current session's user. The sequence of processes that must be performed when accessing the route is given in Figure 34.



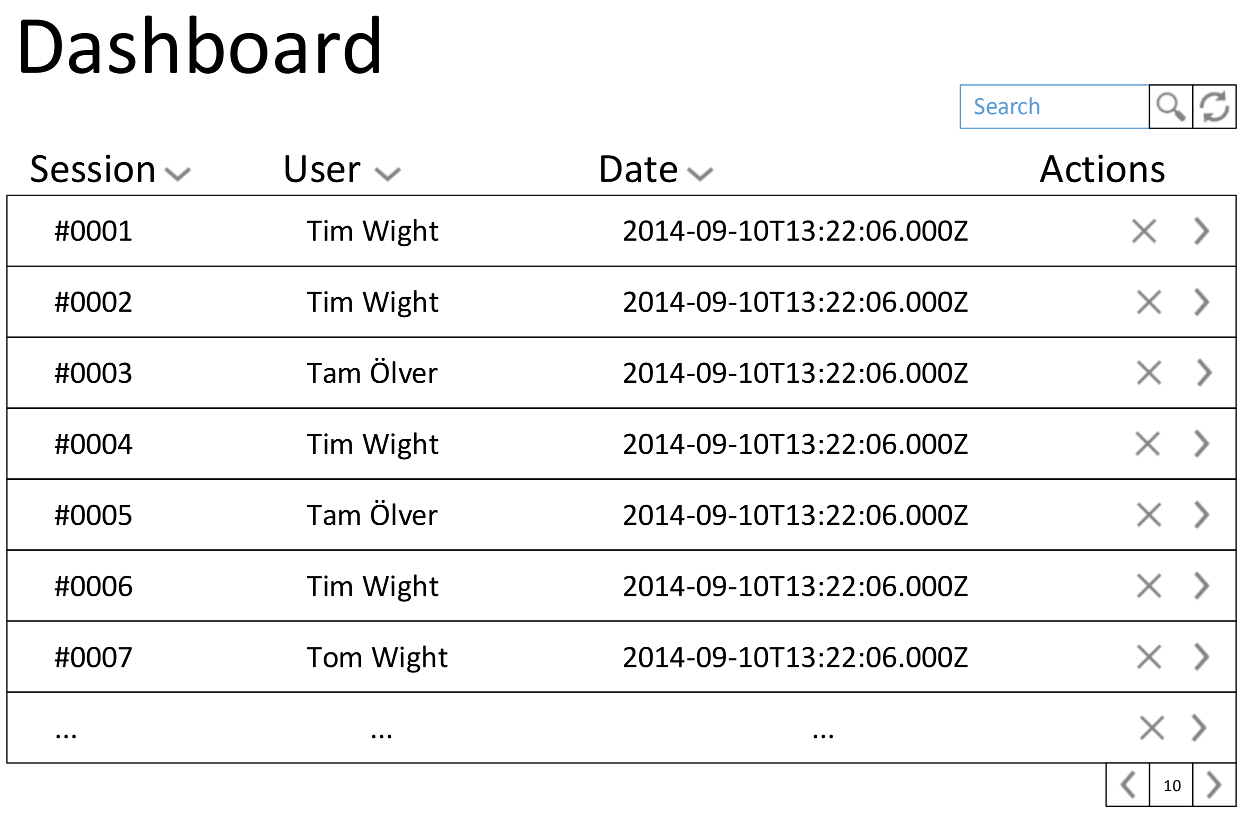
GET http://webserver/session/:uid/:id
Get id and uid params
Generate Jade HTML Index Template
Render HTML
Download JSON from /api/user/:uid
Download JSON from /api/session/:id
Create Chart
Extract timeseries data and critical event data
Create user sessions sidebar
Sort session timeseries data by date
Create chart data series for critical event
All data handled
Remaining data
Populate chart with chart data series'
Populate sidebar with user sessions
Display Page to Client

Figure 34: Web Application Server Session GUI Activity Diagram

6.6.2.3 Graphical User Interface Design

As each of the above routes offer graphical, interactive user interfaces, a GUI design phase was performed in order to determine how best to present the persisted data visually to a user. The result of this analysis determined that a combination of tabular and charted data provided the most intuitive approach to interacting with the data. That is, the index page listing all of the sessions would be best served by listing the data in a tabular format, allowing controls to filter and sort the table data, while the session page would be best if it presented the session data visually using charts, plotting the sensor datapoints against time and highlighting those datapoints that exceeded safe thresholds.

Wireframe designs prototyping this behaviour are given below in Figure 35 and Figure 36. The implementation of these pages use the Twitter Bootstrap CSS library for additional styling, as well as the BootstrapTable plugin by Wen Zhi Xin and the AMCharts plugin by AMCharts.



Session	User	Date	Actions
#0001	Tim Wight	2014-09-10T13:22:06.000Z	× ›
#0002	Tim Wight	2014-09-10T13:22:06.000Z	× ›
#0003	Tam Ölver	2014-09-10T13:22:06.000Z	× ›
#0004	Tim Wight	2014-09-10T13:22:06.000Z	× ›
#0005	Tam Ölver	2014-09-10T13:22:06.000Z	× ›
#0006	Tim Wight	2014-09-10T13:22:06.000Z	× ›
#0007	Tom Wight	2014-09-10T13:22:06.000Z	× ›
...	...	...	× ›

Figure 35: Web Application Server Index GUI Wireframe

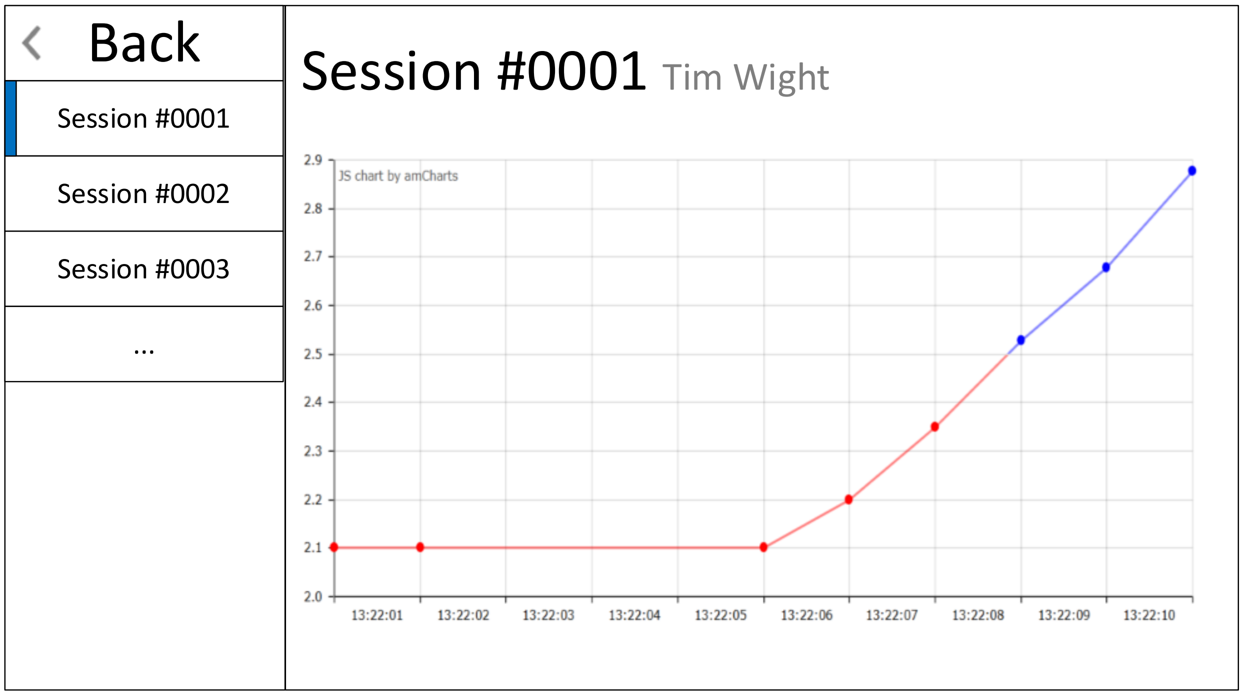


Figure 36: Web Application Server Session GUI Wireframe

6.7 Tools and Vendor Documentation

The development of the system required the use of many tools and platform libraries. Most of these are mentioned throughout the course of this document. We present each of the tools used below for simplicity and clarity. The descriptions given are paraphrased from the vendor documentations themselves.

6.7.1 IntelliJ IDEA

See: https://www.jetbrains.com/idea/documentation

IntelliJ IDEA is a Java IDE by JetBrains with a comprehensive set of tools and integrations with important frameworks and technologies, for both enterprise and web development. It supports a number of languages, such as Java, Scala, Groovy, and many others, and provides support for key frameworks and SDKs such as Android, node.js, and Express.

6.7.2 Android

See: http://developer.android.com

Android is a popular mobile operating system (OS), based on the Linux kernel, developed by Google. It is designed primarily for touchscreen devices such as smartphones and tablet computers, with specialised versions for televisions, cars, and wearable devices such as smartwatches and health bands.

6.7.3 MongoDB

See: http://docs.mongodb.org

MongoDB (from "humongous") is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favour of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. Released under a combination of the GNU Affero General Public License and the Apache License, MongoDB is free and open-source software.

6.7.4 Node.js

See: http://nodejs.org/documentation

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

6.7.5 Express

See: http://expressjs.com/api.html

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Including a myriad of HTTP Utility methods and middleware for simple development of robust web APIs.

6.7.6 MongoHQ/Compose

See: https://docs.compose.io

Compose is a fully-managed online platform usedbydevelopers to deploy, host, and scaledatabases. It provides online MongoDB database provisioning and scaling as well as administrative tools on SSD-backed storage platforms. Included is a free-to-use "sandbox" price tier that allows free use of up to 512 MB of MongoDB data.

6.7.7 Mongoose ODM

See: http://mongoosejs.com/docs

Mongoose is MongoDB object modeling tool written as a Node.js library designed to work an asynchronous environment. In translates between database objects and JavaScript objects for use in JavaScript applications.

6.7.8 Twitter Bootstrap

See: http://getbootstrap.com

Bootstrap is a free collection of tools for creating responsive websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions.

6.7.9 BootstrapTable by Wen Zhi Xin

See: http://wenzhixin.net.cn/p/bootstrap-table/docs/documentation.html

Bootstrap Table is designed to display data in a tabular format and offer rich support for presentational and functional features such as data sorting, pagination, radio and checkbox controls, and so on. The table has been designed to reduce development time and to require no specific knowledge from developers.

6.7.10 AMCharts

See: http://docs.amcharts.com/3/javascriptcharts

amCharts is an advanced charting library for data visualisation. It supports Column, Bar, Line, Area, Step, Step without risers, Smoothed line, Candlestick, OHLC, Pie/Donut, Radar/ Polar, XY/Scatter/Bubble, Bullet, Funnel/Pyramid charts as well as Gauges. A free version as well as a commercial version is available for use.

6.7.11 Stringtree JSON

See: http://www.stringtree.org/stringtree-json.html

Stringtree JSON is a small, neat and robust Java implementation of a reader, writer, and validator for the JSON (JavaScript Object Notation) data format. The reader and writer are one class each, with no dependencies at all.

7 Testing and Results

The system implementation is performed according to the design specification given in the previous sections. Several tests are devised in order to measure the efficiency and effectiveness of the system across several key areas. The primary focus of these tests is to measure the effectiveness of the system design against the core requirements of the problem domain.

The tests, unless otherwise stated, are performed with the client sensor node application operating on a Google Nexus 4 device running Android 4.4.2 and a HTC One M7 running Android 4.4.3 with HTC Sense 6.0, connected via Bluetooth. The server application is hosted locally on a Microsoft Surface Pro 2. All devices in the test configuration were connected on a 2.4 GHz Wi-Fi 802.11 local area network.

As notation convention during the discussion of these tests, we use Delta (Δ) to signify the finite difference or change between two discrete values, delta (δ) to signify a percentage difference or approximation error, and epsilon (ϵ) to give the absolute error of a measurement.

7.1 Quaternion Rotation Efficiency Benchmark

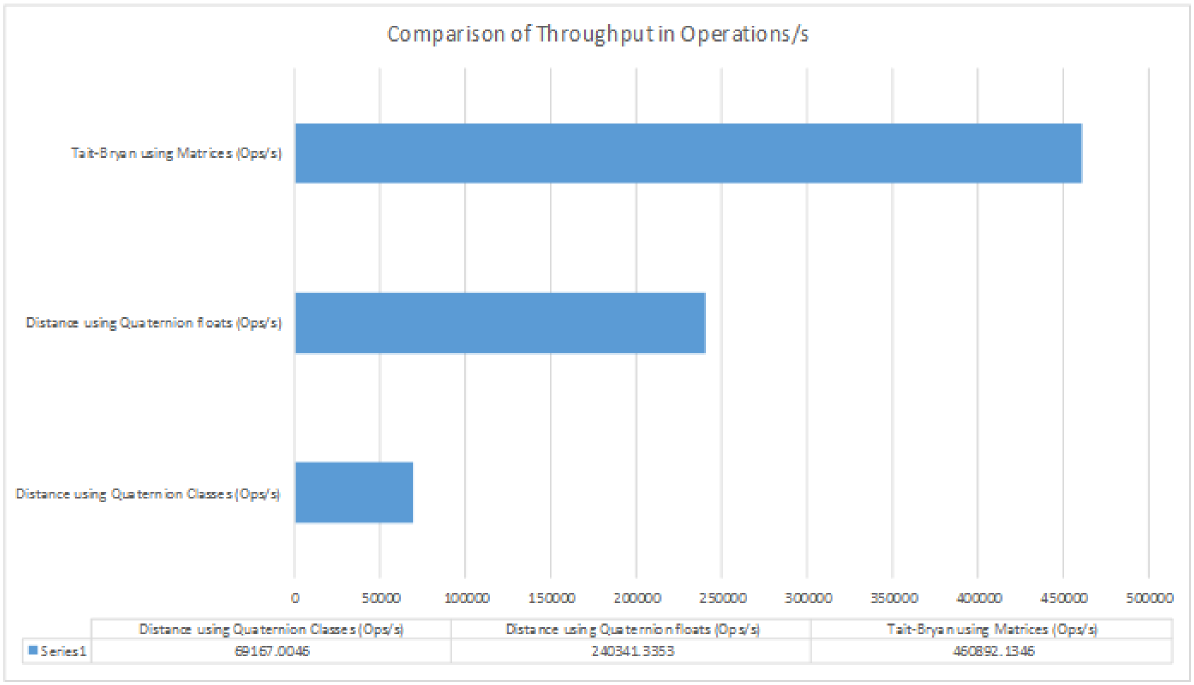
An analysis of two different workloads, each tied to a particular rotation formalism, is conducted; those of the Android-provided matrix operations and of the Quaternion library functions presented in this document. Of the Quaternion functions, two mechanisms are presented for comparison: that is, a distinction is made between using class instantiations, which are easier to use, although bear the cost of heavy memory impact, and using raw floating point values, which are less expensive but more difficult to implement.

The matrix workload computes rotation matrices from two orientation vectors, each expressing the rotation required to transform from the world reference frame to the orientation given by the vectors. From these two rotation matrices, the workload then computes the change in angle using a Tait-Bryan formalism expressed in radians.

The Quaternion workload computes the two orientation vectors as Quaternions, if they are not already expressed as such, then computes the difference Quaternion between them. From the difference Quaternion, the workload then computes its magnitude, giving the magnitude of the flexion/reflexion angle in radians. Note that this workload could be made more efficient by calculating only the distance between the two orientation vectors expressed as Quaternions, but the extra steps were taken to produce the difference Quaternion so that this could later be used to produce an output in Tait-Bryan notation.

The data presented below shows each workload performed on a HTC One M7 at multiples of 100,000 randomly generated vectors. The first dataset gives the total elapsed time (in nanoseconds) of each workload. The second dataset gives the computation efficiency of each workload measured as average throughput. The general formula for calculating throughput is given by Equation 25.

$$Throughput = (Ops/s) = \frac{num\_vectors}{time(ns) \cdot 10^{-9}} \tag{25}$$



	Distance using Quaternion Classes (Ops/s)	Distance using Quaternion floats (Ops/s)	Tait-Bryan using Matrices (Ops/s)
🟦 Series1	69167.0046	240341.3353	460892.1346

Figure 37: Comparison of rotation libraries measured in throughput (operations per second).

As expected, the class-based implementation falls significantly behind. The Android implementation of matrices yields the highest throughput, despite lesser space efficiency. However, it is worth noting that the floating point implementation of the Quaternion library yielded superior throughput when tested on a desktop environment. The reasoning of this discrepancy is uncertain, but is likely due to the difference in CPU architecture. It is, regardless, immediately apparent that each solution is more than adequate to meet the demands of the system.

7.2 Rotation Accuracy

To measure rotational accuracy, several tests are devised. To ensure the accuracy of these, each test configuration is attached to a mechanical pendulum unit with two "limbs" (one fixed, the other the free-moving dual-axis pendulum) simulating the rotational movement of a human joint. By fixing one axis of rotation on the pendulum, the mechanical unit allowed for fine calibration of the magnitude of the angle of rotation across the second axis, henceforth referred to as the flexion/extension axis across the joint. This allows each test to sample its recorded data against the real-world magnitude of the angular distance as measured by the mechanical unit. We will refer to this mechanical unit as the "testing platform", upon which each test is conducted.

7.2.1 Analysis of Sensor Network Rotation Accuracy

This test is intended to give a measure of the accuracy of the rotation sensors in the Android system. The test uses a prototype of the final client sensor network Android application, reconfigured as a more viable testing module. The module pairs with another Bluetooth-enabled Android smartphone, performs a round of time-synchronisation, and begins sampling data from the rotation vector sensor. This data is pipelined to the master node and the angular distance between each device's orientation is calculated using the Quaternion library floating point implementation. Sampling is performed at a rate of 5 Hz (5 samples per second).

The test configuration is arranged so that the master and slave nodes are each attached to a separate limb of the test platform, with each phone facing the same direction; both devices are placed face-up on the same side of the test platform, so that each device is perfectly aligned when the pendulum of the test platform is resting at 0 degrees. The platform is then moved through a differing configurations and the angular distance between each of its limbs is measured manually. At the time of each measurement, the current reading from the master device is recorded. We provide ϵ and δ of the difference between these figures for observation.

Additionally, due to the sensitivity of the magnetometer and its impact on the rotation vector sensor (as this sensor receives the filtered data from the magnetometer and other inertial sensors), we additionally sample the raw magnetometer data from the master and slave devices and compare these to obtain a delta in each axis. The two devices were oriented on the test platform so that they were Z-aligned at all points, while the X- and Y- axes rotated with the motion of the slave device. As such, we can assume the X- and Y- axis magnetometer data to linearly change as the angular distance increases, while the Z-axis should remain constant. Therefore, for the best indicator of magnetic interference affecting measured results, we may consider the

Δ of the Master and Client Z-axis magnetometer data, and hence attempt to correlate it against the epsilon value of the angular distance readings, θ.

Angle	Theta θ	Epsilon ϵ	delta δ	ΔMagX(mT)	ΔMagY(mT)	ΔMagZ(mT)
0.0	2.467579	2.467579	INF	2.470969	4.714962	10.066869
10.0	9.619248	0.380752	3.96%	5.999656	3.600861	8.568239
20.0	18.751280	1.24872	6.66%	14.723745	2.173012	8.964971
30.0	28.949829	1.050171	3.63%	23.159004	0.947389	9.264809
40.0	44.490357	4.490357	10.09%	30.395068	7.534843	13.200798
65.0	64.191609	0.808391	1.26%	48.851187	21.239343	7.872614
90.0	92.798583	2.798583	3.02%	58.871107	38.723234	6.888399

Table 5: Results of Sensor Network Analysis Test, showing the fixed real-world angular distance and the calculated angular distance θ, with X-, Y-, and Z- axis Magnetometer data to present magnetic interference.

The chart below shows a summary of this information by plotting Epsilon (ϵ) against the average delta (Δ) of the Z-Axis of each node's magnetometer data. While not definitive, the plot shows some correlation between changes in the magnetometer (as fluctuating magnetic fields influence its readouts) and the changes in epsilon of the angular distance calculations.

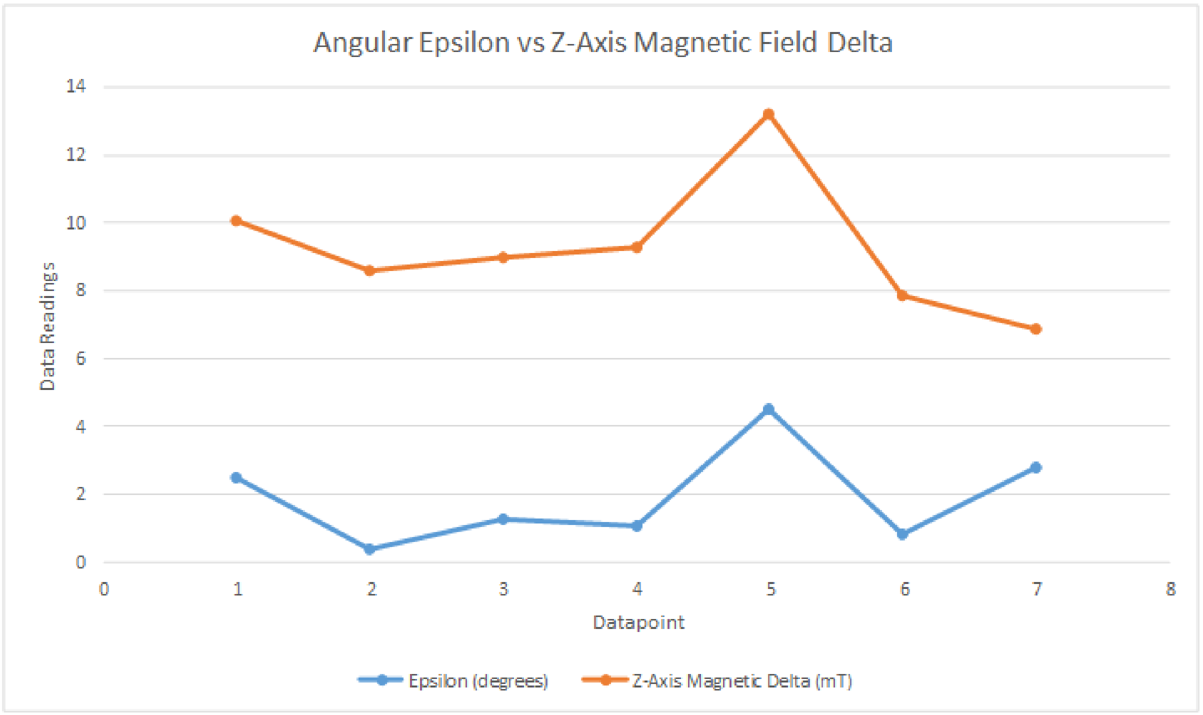


Figure 38: Correlation between angular epsilon vs Z-Axis magnetic field Delta as indication of the effects of magnetic interference on the accuracy of the system results.

From the results of this test, we can see that the system experiences an average absolute error of 1.892079 degrees in a moderately controlled environment, or a percentage error of between 1.26 and 10.09 percent.

7.2.2 Analysis of Sensor Drift

The next test is devised to provide an approximation of the effect of drift error in the inertial sensors. The test is configured identically to the previous, with two Android devices attached to the test platform and set to a fixed angular distance from each other of 0 degrees. The system then begins sampling data and an initial measurement of angular distance is produced by the system. An attempt is then made to introduce drift error to the client node by exerting significant rotational and linear acceleration on the device before restoring it to the previous position of 0 degrees. The angular distance as measured by the system is then sampled over time until the measured angle settles to within an acceptable margin of the initial measurement.

The data was sampled at a rate of 5 Hz (200 ms) over a period of approximately 2 minutes and 30 seconds. Drift error was introduced to the client node at $t_1 = 4.6$ seconds until $t_2 = 41.4$ seconds. The average measured distance between time of commencement, t_0, and t_1 gives 2.155562386 degrees. At time t_2 when the test configuration was restored to a real-world 0 degrees, the system read a value of 8.040107, which fluctuated to a maximum of 14.389446 at $t = 1$ minute 18.4 seconds before gradually settling and reaching a minimum of 4.358923 at $t_{final} = 2$ minutes 36.2 seconds when the test was concluded. The graph displaying this data is shown below.

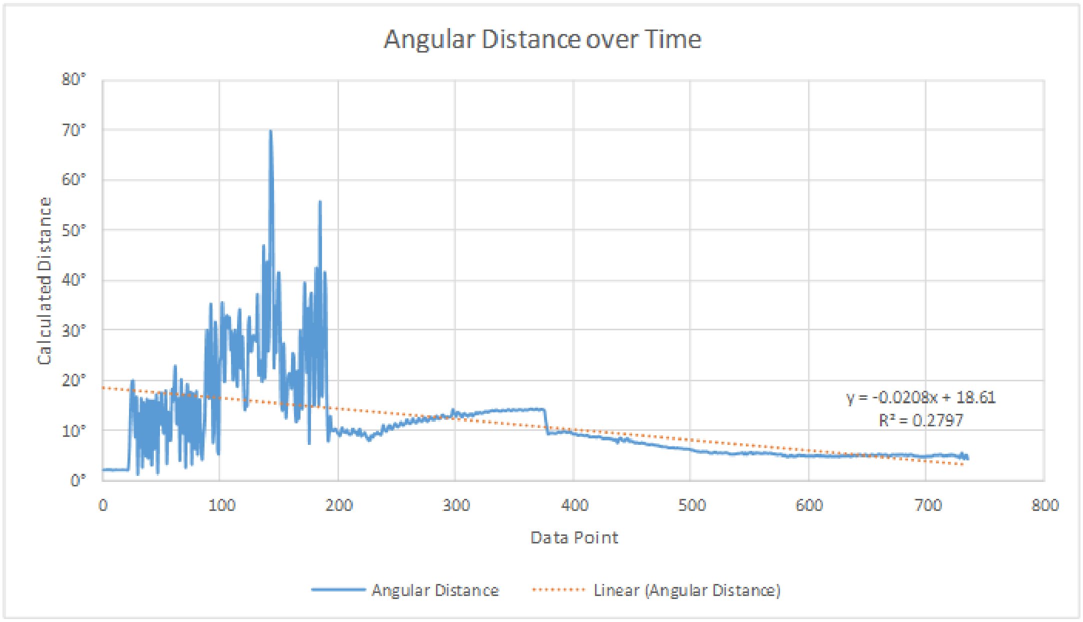


Figure 39: Angular estimation error over time after sensor drift has been introduced to the system.

Basic linear regression models yield a linear trend line of the data points suggesting the average decline in error introduced by drift is given as $y = -0.0208x$, where y is the change in error given in degrees, and x is the period of time since cessation of motion, given as a multiple of 200 ms. That is, in this test case, the drift error declines by approximately 0.0208 degrees per 200 ms, or 0.104 degrees per second. Further experimentation and more exacting test platforms are required to prove this result more rigorously, but this model yields serviceable insight into the expected error of the system during periods of increased motion and activity.

8 Future Development

The presented system provides a proof-of-concept of an Android-based sensor network to monitor relative angular difference between each node. From the results and discussion of the system we can see the described system carries merit. However, in continuing development of the system, there are several key directions of development that may be taken to improve the results.

8.1 Data Persistence and Analytics using Big Data Techniques

The current system uses data packets to facilitate communicating available sensor data to the server, which are sent at discrete intervals and contain a finitely large number datapoints. These are then saved to a document-oriented database that saves the packets of data as-is. This process results in chunked data that is not immediately available, and may take some period of time to become available. It also means that any missing fields from the sensor data, particularly calculated fields, must be calculated and inserted to the data either on-receive by the server or on-retrieve by the client, which may induce substantial penalties in calculation times based on the size of the dataset. Any future analytics that must touch every datapoint in a session are also liable to induce similar costs.

Using batch processing techniques such as MapReduce or equivalent technologies in the Hadoop ecosystem, this process may be performed in a scheduled workload intended to produce and persist the required analytic data. Future requests for analytic data would then have this available. However, this requires that the input data be finalised before computation, and therefore requires the capturing session be concluded, or else require that the job be rerun multiple times.

To circumvent this requirement, we may introduce alternative technologies such as Storm or Spark. By modifying the current server implementation to use socket-based IO, we can enable a data streaming protocol between client sensor networks and the server, as opposed to using packet-based protocols. This means datapoints can be made instantly available for review on the server. Combining this with Big Data technologies suitable for realtime streaming data such as Spark, we can calculate analytic data on-the-fly as the data is received, and hence review realtime information concerning a data capture session.

Note that in implementing this solution, some modifications should be made to the existing database and server implementations. Most notably, they must each be configured for "cloud"-oriented use. In MongoDB (the data persistence engine currently employed), this can be achieved via "Sharding", in which a data collection can be partitioned across multiple nodes via a “shard key”, where each node may represent a separate machine. The web application, that receives incoming requests and handles data persistence and retrieval from the database shards, can have multiple instance running on separate machines if we introduce what is known as a “load balancer”–a separate machine that serves as a port of call for all incoming requests, then forwards these requests to one of the server instances based on which server is currently the most available. These servers may then use another cluster of machines configured as, for example, a Spark cluster, to analyse incoming session data and calculate additional fields such as angles of reflexion/extension, or perform any other analytical requirements.

8.2 Bluetooth Low Energy and Wearable Technologies

An emerging technology at the time of writing exists in what is known as “Wearables”; devices such as smart watches and health bands that leverage the low power consumption of BLE (as discussed in subsection 3.2) as well as the available power of smartphones and discrete sensors. Wearable technologies typically aim to augment the existing capabilities of smartphones by offering a “wearable” interface for viewing messages and taking phone calls (as in the case of most smartwatch devices), though many may offer their own capabilities, such as in heart rate and body temperature monitoring. A version of Android, known as Android Wear, is available for use in developing for such Wearable smart devices.

It is therefore not only possible to extend the system prototype to utilise Bluetooth Low Energy in order to reduce power consumption and improve general performance, but may also be beneficial to target wearable devices with the system software rather than smartphones in order to reduce the size, weight, and cost of the system. Though, at the time of writing, there are limited wearable devices that come equipped with the required sensors, this market is expected to grow, and it may be possible to utilise health bands to record the required data less intrusively.

Alternatively, it would also be possible to engineer a custom “wearable” device that runs the Android Wear OS, equipped with the requisite sensors.

8.3 Customised Fused-Sensor Software Filter

The system prototype uses the Android virtual Rotation Vector Sensor, a fused-sensor that filters gyroscopic, accelerometric, and magnetometric sensor data to provide an estimated orientation vector as a “best-guess” of the combined inputs. It may therefore be possible to replace this Android filter with a customised filter tailored towards the system domain problem.

In devising such a filter, we may assume that the sensor network is configured such that we can expect a particular range of motion across certain axes. For example, on a human leg we may assume the relative movement across the coronal plane (that is, the plane that divides the body into the dorsal and ventral [front and back]) will be minimal, and therefore define an expected deviation of only a few degrees difference between each sensor placed on the leg. From this, we may define thresholds for the difference in magnetometric and gyroscopic data across the coronal plane such that the filter favours data from the sensors that do not exceed this threshold. We may assume that, in the case of the gyroscopic data exceeding its threshold, an irreconcilable level of drift has been introduced; or, in the case of the magnetometer, that unjustifiable magnetic interference has disproportionately befouled the magnetometric orientation of one or more of the devices (as in the case of all the sensors in the network being equally afflicted, the relative difference between them can still be assumed to be accurate).

We may also offer lesser preference towards accelerometric data, as this is less likely to yield consistent orientation data, and instead use this only to justify gyroscopic drift.

8.4 High-Accuracy Sensor "Symmetry"

Improving the accuracy of the hardware sensors in each node is likely to dramatically increase the accuracy of the complete sensor network. More precise magnetometers will be able to give more precise measurements of their relative alignment to the world magnetic reference frame, while higher precision accelerometers and gyroscopes will assist in minimising errors in drift.

Similarly, by introducing sensor "symmetry" into the network (that is, by minimising the hardware differences between each sensor node in the sensor network), we can minimise the differences in sensor output between each node. For example, suppose that two different magnetometers with varying sensitivities, when aligned to point directly North, have an average divergence in their readings of 1 degree; then we can assume for all readings of the relative difference in orientation between the two devices, we will suffer at least a degree of inaccuracy. Similarly, different hardware gyroscopes may yield different readings for a particular rotation, gradually putting them out of sync in their calculated orientations. Minimising this error will therefore improve overall accuracy.

8.5 Maturation of the Data-Processing Framework

While the data-processing framework presented as part of this prototype adequately provides the structure required to organise data processing logic into components that reflect data flows, it is lacking in terms of higher-level abstractions targeted at easing management and coordination of these components.

As such, pipeline- and components-management forms a large responsibility that is not catered for by the framework and necessitates the existence of a large, overarching coordination entity within the application layer. In the case of the system prototype, this results in a significant portion of the code within the coordinating services layer. For simple pipeline sections, such as the slave coordinating service, this responsibility was manageable (see subsection C.1), however for the complex pipeline implemented by the master coordinating service, the code required for coordination was very significant (see subsection C.2).

Additionally, execution of long pipeline portions of directly-communicating individual components such as data transforms results in large call-stacks due to the nature of each component's atomic operation. Reminiscent of recursive programming, the general form of a data transform call stack looks as follows:

```
(-> signifies reference to next data component)

Given the pipeline: Component A -> Component B -> Service C -> Component C

Calling environment generates data for pipeline
Calling environment invokes onData on first pipeline component
   Component A: onData() executes
   Component A does processing
   Component A invokes onData() of attached IDataListener
      Component B: onData() executes
      Component B does processing
      Component B invokes onData() of attached IDataListener
         Service C: onData() executes
         Service C does processing
         Service C invokes onData() of attached IDataListener
            Component D: onData() executes
            Component D does processing
            Component D does not have an attached IDataListener
            Component D: onData() returns
         Service C: onData() returns
      Component B: onData() returns
   Component A: onData() returns
Control returned to calling environment
```

The framework would greatly benefit from the inclusion of additional utilities that would form component containers, to which the responsibilities of managing and executing individual low-level components could be delegated.

Furthermore, the implementation of parsing techniques and Java reflection could be leveraged to extend the framework to incorporate data-driven ideologies such as the definition of data pipeline structures in external data files to be loaded at runtime. These data files would consist of a series or flow of data-processing component identifiers and parameters that correspond to pre-defined components (see paragraph 6.6.1.3), and would remove the need for inflexible, compile-time pipeline definitions in code. This would further reduce the responsibilities of application-level code, and also reduce the focus of data processing code overall. Further, it promotes the pipeline-oriented objectives and flexibility of the framework.

8.6 Adaptive Sensor Sampling Rates

The developed system prototype collects the desired sampling frequency for the sensors, measured in Hertz, at commencement of a sampling session. This value then remains fixed for the duration of the session. As has been discussed in subsection 3.1, drift error in inertial sensors occurs as a result of discontinuous sensor sampling (that is, we can only sample data at discrete intervals). Hence, during periods of high inertial activity, the sensors accumulate a significant level of drift errors due to the low sampling rate.

A proposed solution would involve adaptive sensor sampling rates based on current levels of motion. During periods of high inertial activity, when the readings from the accelerometer or gyroscope indicate significant change in forces, we may request the sensors sample at a higher rate in order to mitigate the inherent drift errors. During times of low activity, we may seek to conserve power and space by performing the opposite.

This requires significant work inn synchronised communication between the master and slave sensor nodes in order to ensure each device's sensors remain synchronised with the other.

A Appendix

Provided here is the code executed to produce the provided results given in the Quaternion Rotation Efficiency Benchmark, followed by the raw output results of the benchmark.

A.1 Quaternion Rotation Efficiency Benchmark Code Sample

```
// Distance using Quaternion Classes
t0 = System.nanoTime();
for (int i = 1; i < num; ++i)
{
    q1 = new Quaternion(vecs[i-1]);
    q2 = new Quaternion(vecs[i]);
    qDiff = q1.Difference(q2);
    flexion_delta = qDiff.Magnitude();
}
t1 = System.nanoTime();

qcTime = t1 - t0;

// Distance using Quaternion Floating Points
t0 = System.nanoTime();
for (int i = 1; i < num; ++i)
{
    Quaternion.getDifferenceQuaternion(qfDiff, vecs[i-1], vecs[i]);
    flexion_delta = Quaternion.getQuaternionMagnitude(qfDiff);
}
t1 = System.nanoTime();

qfTime = t1 - t0;

// Distance using Rotation Matrices
t0 = System.nanoTime();
for (int i = 1; i < num; ++i)
{
    RotationMatrix.getRotationMatrixFromVector(r1, vecs[i - 1]);
    RotationMatrix.getRotationMatrixFromVector(r2, vecs[i]);
    RotationMatrix.getAngleChange(taitbryan_delta, r1, r2);
}
t1 = System.nanoTime();
```

```
mfTime = t1 - t0;
```

A.2 Rotation Formalism Efficiency Benchmark Raw Data

# vectors	Distance using Quaternion Classes (ns)	Distance using Quaternion floats (ns)	Tait-Bryan using Matrices (ns)
100000	1374075995	399865710	237063940
200000	3026354341	891512284	447617885
300000	4465429574	1274988554	625386845
400000	5755803448	1611213184	841464977
500000	7112894856	2051463451	1040021364

Table 6: The average time in nanoseconds taken across ten samples to complete each vector workload.

A.3 Rotation Formalism Throughput Raw Data

# vectors	Distance using Quaternion Classes (Ops/s)	Distance using Quaternion floats (Ops/s)	Tait-Bryan using Matrices (Ops/s)
100000	72776.17858	250083.9594	421827.1239
200000	66086.11467	224337.9072	446809.6712
300000	67182.78612	235296.23	479703.0868
400000	69495.07634	248260.1334	475361.4362
500000	70294.8673	243728.4465	480759.3549
Average	69167.0046	240341.3353	460892.1346

Table 7: The average throughput of each rotation class formalism.

B Appendix

Provided here is the code executed to produce the provided results given in the Rotation Accuracy tests, followed by the raw output results of the benchmark.

B.1 Orientation Accuracy Test Code Sample

```
// Aggregate the master and slave sensor data
AggregatorDataTransform aggregator = mAggregators.get(index);
Data aggregatedData = aggregator.transform(data);
aggregatedData.set(mIntervalKey, mTimeOffset + (timeslice * mIntervalTime));

...

// Calculate the flexion/reflexion anglular magnitude
prevQuaternion = data.get(prevFields);
newQuaternion = data.get(newFields);
Quaternion.getDifferenceQuaternion(diffQuaternion, newQuaternion, prevQuaternion);
float magnitude = Quaternion.getQuaternionMagnitude(diffQuaternion);
magnitude = (float)Math.toDegrees(magnitude);

// Save difference quaternion
data.set(mDifferenceFields, diffQuaternion);
// Save magnitude
data.set(mMagnitudeField, magnitude);
```

B.2 Orientation Accuracy Test Raw Data

Real (°)	System Est. (°)	Master Mag X (mT)	Master Mag Y (mT)	Master Mag Z (mT)	Slave Mag X (mT)	Slave Mag Y (mT)	Slave Mag Z (mT)
0	2.501766	-9.118652	15.658569	58.198547	-6.779999	11.16	48.42
0	2.489573	-9.059143	15.658569	58.259583	-6.96	10.679999	48.12
0	2.469082	-9.239197	15.658569	58.198547	-6.48	11.04	48.059998
0	2.454146	-9.239197	15.539551	58.31909	-6.54	10.92	48.539997
0	2.423326	-9.118652	15.539551	58.198547	-6.66	10.679999	47.7
10	9.632857	6.3598633	54.899597	-9.779358	0.48	58.739998	-18.3
10	9.623027	6.4193726	54.899597	-9.719849	0.84	58.26	-18.24
10	9.601282	6.3598633	55.018616	-9.899902	-0.179999	58.8	-18.42
10	9.613153	6.4193726	55.13916	-9.719849	0.06	59.28	-18.72
10	9.625919	6.1798096	55.378723	-9.719849	0.539999	58.26	-18
20	18.764217	5.999756	54.838562	-10.319519	-8.88	57.239998	-19.26
20	18.73618	5.999756	54.899597	-10.379028	-8.82	56.399998	-19.8
20	18.784906	5.999756	54.899597	-10.319519	-8.639999	56.82	-19.199999
20	18.75157	5.999756	55.018616	-10.438538	-8.88	57.239998	-19.199999
20	18.719528	5.819702	54.838562	-10.438538	-8.58	57.66	-19.26
30	28.874102	5.8792114	55.259705	-10.85968	-17.64	53.82	-19.619999
30	28.975985	5.999756	55.259705	-10.679626	-17.22	54.48	-19.859999
30	28.859295	5.9387207	55.378723	-10.559082	-17.34	54.18	-20.279999
30	29.071918	5.758667	55.07965	-10.438538	-16.98	54.059998	-19.5
30	28.967844	5.758667	55.13916	-10.379028	-17.279999	54.84	-19.98
40	44.534195	5.758667	55.378723	-10.499573	-24.119999	47.46	-23.939999
40	44.481865	5.8792114	55.378723	-10.618591	-24.6	48	-24
40	44.508057	6.478882	55.558777	-10.85968	-23.939999	48.3	-23.279999
40	44.443493	6.239319	55.378723	-10.559082	-24.539999	48.18	-23.58
40	44.484177	6.059265	55.499268	-10.559082	-24.359999	47.579998	-24.3
90	92.83589	7.0785522	55.558777	-11.399841	-51.539997	17.1	-18.24
90	92.77845	7.1395874	55.799866	-11.51886	-51.36	17.16	-19.02
90	92.76042	7.0785522	55.73883	-11.759949	-51.84	16.98	-18.3
90	92.774345	7.319641	55.799866	-11.579895	-51.78	16.68	-18.539999
90	92.84381	7.559204	55.73883	-11.819458	-51.66	17.1	-18.42
65	64.22264	6.1187744	55.73883	-10.85968	-42.78	34.62	-19.08
65	64.234955	6.1187744	55.67932	-10.798645	-42.36	34.26	-18.3
65	64.12225	6.298828	56.03943	-10.919189	-42.3	34.5	-18.84
65	64.20145	5.999756	55.799866	-10.85968	-42.719997	34.68	-18.72
65	64.17675	6.1798096	55.499268	-11.039734	-43.379997	34.5	-18.9

Table 8: The raw sensor output sampled across 5 datapoints through 7 different angular configurations.

C Appendix

C.1 Slave Pipeline Creation Code Sample

Provided here is the code responsible for the creation of the data-processing components that form the slave portion of the data pipeline in the system.

```
...

// Retrieve relevant configuration
String[] sensorKeys = mSensorConfig.get("sensorKeys");
String[] splitKeys = { "X", "Y", "Z", "W" };

// Create array splitter each for sensor type
DataTransform[] transforms = new DataTransform[sensorKeys.length];
for(int i = 0; i < sensorKeys.length; ++i)
{
   transforms[i] = new ArraySplitDataTransform(sensorKeys[i], splitKeys, true);
   if(i > 0)
      transforms[i-1].setDataListener(transforms[i]);
}

// Integrate data services at beginning and end
mSensorService.setDataListener(transforms[0]);
transforms[transforms.length-1].setDataListener(mBluetoothService);

...
```

C.2 Master Pipeline Creation Code Sample

Provided here is the code responsible for the creation of the data-processing components that form the master portion of the data pipeline in the system.

```
...

// Slave pipeline components from bluetooth to aggregation
ArithmeticDataTransform timestampCorrector = new ArithmeticDataTransform(
   KEY_TIMESTAMP, mBluetoothService.getTimeOffset(),
   ArithmeticDataTransform.Operation.Subtract);
FieldCopyDataTransform slaveTimestampCopy = new FieldCopyDataTransform(
   KEY_TIMESTAMP, KEY_TIMESTAMP_COPY);
String[] timestampCopyField = { KEY_TIMESTAMP_COPY };
FieldModifyDataTransform slavePrepend = new FieldModifyDataTransform(
   KEY_SLAVE_PACK, timestampCopyField, true, true);
String[] timestampField = { KEY_TIMESTAMP };
PackDataTransform slavePack = new PackDataTransform(KEY_SLAVE_PACK,
   timestampCopyField);
FieldRenameDataTransform slaveTimestampRestore =
   new FieldRenameDataTransform(timestampCopyField, timestampField);

// Create array splitters for master sensor data
String[] sensorKeys = mConfig.get(SensorSampleService.CONFIG_SENSOR_KEYS);
String[] splitKeys = { "X", "Y", "Z", "W" };
DataTransform[] arraySplits = new DataTransform[sensorKeys.length];
for(int i = 0; i < sensorKeys.length; ++i)
   arraySplits[i] = new ArraySplitDataTransform(sensorKeys[i], splitKeys, true);

// Master pipeline from array splits to aggregation
FieldCopyDataTransform masterTimestampCopy = new FieldCopyDataTransform(
   slaveTimestampCopy);
FieldModifyDataTransform masterPrepend = new FieldModifyDataTransform(
   KEY_MASTER_PACK, timestampCopyField, true, true);
PackDataTransform masterPack = new PackDataTransform(KEY_MASTER_PACK,
   timestampCopyField);
FieldRenameDataTransform masterTimestampRestore = new FieldRenameDataTransform(
   slaveTimestampRestore);

// Aggregator component
String[] aggregatorKeys = {KEY_SLAVE_PACK, KEY_MASTER_PACK};
// 1000ms divided by the sampleRate = intervalTime
int intervalTime = 1000 / (Integer)mConfig.get(
   SensorSampleService.CONFIG_SAMPLE_RATE);
mAggregator = new IntervalAggregatorDataTransform(KEY_TIMESTAMP, intervalTime,
   mStartTime, aggregatorKeys);

// Pipeline components after aggregation until after data persistence
UnpackDataTransform slaveUnpack = new UnpackDataTransform(KEY_SLAVE_PACK);
UnpackDataTransform masterUnpack = new UnpackDataTransform(KEY_MASTER_PACK);
SetDataTransform setSessionId = new SetDataTransform(KEY_SESSION, mSessionId);
String[] masterFields = { "masterRotDataX", "masterRotDataY", "masterRotDataZ",
   "masterRotDataW"};
String[] slaveFields = { "slaveRotDataX", "slaveRotDataY", "slaveRotDataZ",
   "slaveRotDataW" };
String[] differenceFields = { "calcDiffOrientationX", "calcDiffOrientationY",
   "calcDiffOrientationZ", "calcDiffOrientationW"};

QuaternionDifferenceDataTransform kneeAngle = new
   QuaternionDifferenceDataTransform(masterFields, slaveFields,
   differenceFields, KEY_ORIENTATION_DIFF);
RemoveDataTransform removeSessionId = new RemoveDataTransform(KEY_SESSION);
SetDataTransform setSessionId2 = new SetDataTransform(KEY_SESSION, mSessionId);
mArrayCollector = new ArrayCollectDataTransform(mArrayCollectCount,
KEY_ARRAY_COLLECT);
SetDataTransform setUserId = new SetDataTransform(KEY_USER, mUserId);
Data[] events = new Data[] { mEventData };
SetDataTransform setEventData = new SetDataTransform(CONFIG_EVENT_DATA, events);
// Create the broadcast data source
mBroadcastSource = new BroadcastDataSource();

// Thread and handler to handle data as soon as possible
mSlaveToThreadHandler = new DataEventHandler(null, mThread.getLooper());
mMasterToThreadHandler = new DataEventHandler(null, mThread.getLooper());
// Handler to allow data to return to UI thread at completion of pipeline
mPersistFromThreadHandler = new DataEventHandler(null, getMainLooper());
mRemoteFromThreadHandler = new DataEventHandler(null, getMainLooper());

// Set the pipeline - slave until aggregator
mBluetoothService.setDataListener(mSlaveToThreadHandler);
mSlaveToThreadHandler.setDataListener(timestampCorrector);
DataTransform.pipeline(timestampCorrector, slaveTimestampCopy, slavePrepend,
   slavePack, slaveTimestampRestore, mAggregator);

// Set the pipeline - Master until aggregator
mSensorService.setDataListener(mMasterToThreadHandler);
mMasterToThreadHandler.setDataListener(arraySplits[0]);
DataTransform.pipeline(arraySplits);
DataTransform.pipeline(arraySplits[arraySplits.length-1], masterTimestampCopy,
   masterPrepend, masterPack, masterTimestampRestore, mAggregator);

// Aggregator to broadcast to master (via handler attached back to main thread)
DataTransform.pipeline(mAggregator, slaveUnpack, masterUnpack, setSessionId,
   kneeAngle);
kneeAngle.setDataListener(mDataStoreService);
mDataStoreService.setDataListener(mBroadcastSource);
mBroadcastSource.setDataListener(mPersistFromThreadHandler);
// Broadcast to master via remoteService
mBroadcastSource.setDataListener(removeSessionId);
DataTransform.pipeline(removeSessionId, mArrayCollector, setSessionId2,
   setUserId, setEventData);
setEventData.setDataListener(mRemoteService);
mRemoteService.setDataListener(mRemoteFromThreadHandler);

mPersistFromThreadHandler.setDataListener(this);
mRemoteFromThreadHandler.setDataListener(this);

...
```

D Appendix

All source code is available under the MIT License from:

https://github.com/sampsonjoliver/latrobe-datacapture-dir

References

[1] Vicon. T-series optical tracking system. Vicon. [Online]. Available: http://www.vicon.com/System/TSeries

[2] MetaMotion. System motion capture system. MetaMotion. [Online]. Available: http://www.metamotion.com/gypsy/gypsy-motion-capture-system.htm

[3] R. Williamson and B. Andrews, "Detecting absolute human knee angle and angular velocity using accelerometers and rate gyroscopes," *Medical and Biological Engineering and Computing*, vol. 39, no. 3, pp. 294–302, 2001. [Online]. Available: http://dx.doi.org/10.1007/BF02345283

[4] D. Roetenberg, *Inertial and Magnetic Sensing of Human Motion.* University of Twente [Host], 2006. [Online]. Available: http://books.google.com.au/books?id=7Y2qtgAACAAJ

[5] D. Vlasic, R. Adelsberger, G. Vannucci, J. Barnwell, M. Gross, W. Matusik, and J. Popović, "Practical motion capture in everyday surroundings," *ACM Trans. Graph.*, vol. 26, no. 3, Jul. 2007. [Online]. Available: http://doi.acm.org/10.1145/1276377.1276421

[6] H. Zhou, T. Stone, H. Hu, and N. Harris, "Use of multiple wearable inertial sensors in upper limb motion tracking," *Medical Engineering and Physics*, vol. 30, no. 1, pp. 123–133, 2006. [Online]. Available: http://www.medengphys.com/article/S1350-4533(06)00263-3/abstract

[7] M. El-Gohary, S. Pearson, and J. McNames, "Joint angle tracking with inertial sensors," in *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, Aug 2008, pp. 1068–1071.

[8] P. S. Maybeck, "The kalman filter: An introduction to concepts," in *Autonomous Robot Vehicles.* Springer, 1990, pp. 194–204.

[9] R. Balani, "Energy consumption analysis for bluetooth, wifi and cellular networks," *[Online]. http://nesl.ee.ucla.edu/fw/documents/reports/2007/PowerAnalysis.pdf*, 2007. [Online]. Available: http://nesl.ee.ucla.edu/fw/documents/reports/2007/PowerAnalysis.pdf

[10] M. Ringwald and K. Romer, "Practical time synchronization for bluetooth scatternets," in *Broadband Communications, Networks and Systems, 2007. BROADNETS 2007. Fourth International Conference on*, Sept 2007, pp. 337–345.

[11] J.-S. Lee, Y.-W. Su, and C.-C. Shen, "A comparative study of wireless protocols: Bluetooth, uwb, zigbee, and wi-fi," in *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, Nov 2007, pp. 46–51.

[12] J. Lin and K. Nikita, *Wireless Mobile Communication and Healthcare: Second International ICST Conference, MobiHealth 2010, Ayia Napa, Cyprus, October 18 - 20, 2010. James C. Lin ; Revised Selected Papers*, ser. Lecture notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering: Institute for

Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2011. [Online]. Available: http://books.google.com.au/books?id=b1x17t2SeKkC

[13] B. Redding and Q. Atheros, Eds., *Bluetooth Technology - Advanced*, ser. Bluetooth SIG. Bluetooth World, 2013.

[14] Bluetooth, "Core version 4.1," Bluetooth Special Interest Group, Tech. Rep., December 2013. [Online]. Available: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=282159

[15] Cisco, "802.11ac: The fifth generation of wi-fi," Cisco, Tech. Rep., August 2012. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/wireless/aironet-3600-series/white_paper_c11-713103.pdf

[16] Litepoint, "Bluetooth low energy," Litepoint, Tech. Rep., 2012. [Online]. Available: http://www.litepoint.com/wp-content/uploads/2014/02/Bluetooth-Low-Energy_WhitePaper.pdf

[17] D. Mills, "Modelling and analysis of computer network clocks," *Electrical Engineering Department Report*, vol. 9252, 1992.

[18] D. L. Mills, "Improved algorithms for synchronizing computer network clocks," *IEEE/ACM Trans. Netw.*, vol. 3, no. 3, pp. 245–254, Jun. 1995. [Online]. Available: http://0-dx.doi.org.alpha2.latrobe.edu.au/10.1109/90.392384

[19] J. Whslén, I. Orhan, and T. Lindh, "Local time synchronization in bluetooth piconets for data fusion using mobile phones," in *Body Sensor Networks (BSN), 2011 International Conference on*, May 2011, pp. 133–138.

[20] Google. Android developer documentation. Google Inc. [Online]. Available: http://developer.android.com/index.html

[21] ——. Android source documentation. Google Inc. [Online]. Available: https://source.android.com/index.html

[22] P. Berner, R. Toms, K. Trott, F. Mamaghani, D. Shen, C. Rollins, and E. Powell, "Technical concepts orientation, rotation, velocity and acceleration, and the srm," *[Online]. http://www.sedris.org/wg8home/Documents/WG80485.pdf*, June 2008, copyright 2008 SEDRIS. [Online]. Available: http://www.sedris.org/wg8home/Documents/WG80485.pdf

[23] J. Diebel, "Representing attitude: Euler angles, unit quaternions, and rotation vectors," *Matrix*, vol. 58, pp. 15–16, 2006.