LONDON'S GLOBAL UNIVERSITY

UCL

Tracking Wildlife Counts Using the Internet Of Things

Matthew Bell[1]
m.bell@cs.ucl.ac.uk

BSc Computer Science

Supervisor: Dr Kevin Bryson

Submission date: 30th April 2018

[1]**Disclaimer:** This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Conservation experts, park rangers, and biologists frequently aim to try to track the location and count of various species of animals for a number of reasons, such as preventing illegal poaching and hunting, monitoring biodiversity, and analysing migration patterns. This is, more often than not, extremely time consuming, since researchers have to install camera traps with motion sensing shutters, and manually look back through images to identify and count the animals.

This project and report explores the possibility of using a low-power computer with sensors, connected to a web server over a wireless Internet connection (a paradigm frequently referred to as *The Internet of Things (IoT)*) to automate this task to save researchers many hours of time when conducting studies using camera traps.

The project also explores various methods in which species of animals could be identified automatically using deep learning techniques, such as using convolutional neural networks, with the constraint of having to operate on remote, low-powered hardware with limited Internet connectivity.

The project proved to be a lot more difficult than first anticipated, thanks to unforseen issues with the hardware being used, as well as difficulties in developing a rigourous image classifier. However, it proved to be an incredibly valuable learning experience, and provided the opportunity to work in a problem domain that would not normally be possible in an undergraduate computer science degree.

Contents

Chapter 1

Introduction and background

1.1 Motivation and need

Wildlife conservation experts constantly need to keep track of the location and movements of wildlife for a number of reasons, including monitoring species migration patterns and population counts [18]. Other options exist, like line transect surveys (counting animals or traces of animals, like tracks and droppings) or track surveys (physically visiting the area and counting animals); but in the comparison study by Silveira et al [30], they found that camera traps, despite their longer initial setup time and cost, "can be handled more easily and with relatively [lower] costs in a long term run".

Based on this, it would be logical to assume that running costs and analysis time could be reduced further by automating the classification of photos taken by camera traps and sending the results back to a web server. This would enable research teams to store, access, analyse and visualise wildlife counts with ease, using an Application Programming Interface (API) provided by the web service.

The biggest advantage of automating the classification stage of camera trap studies is that it would save a lot of time after the main study has ended, and results can be analysed as soon as possible.

1.2 Requirements

A list of requirements for the solution were reasonably easy to devise. Most of the requirements are defined by the limits of the environments where this system may be deployed, such as woodlands, grasslands, and national parks.

Most of the locations where this solution could be deployed may have very limited cell network coverage, and definitely would not have WiFi connectivity available. Therefore, the system would have to use LoRaWAN, which has a theoretical range of up to twenty kilometres. However, the lower data rate of LoRaWAN means that photos captured by the camera traps would not be able to be sent back to a web server, so any kind of image processing and classification would have to be performed on-device.

Another limitation is the devices being used for the project. The main "base station" device is the Creator Ci40 developer board, designed to "allow developers to rapidly create connected products" [23]. This ability to rapidly prototype on the board, which has a -based processor and runs the OpenWRT Linux distribution. It also contains a WiFi radio, useful for communicating with the device and debugging code on it during development, and a 6LoWPAN radio, useful for communicating with nearby devices.

The sensor devices were also provided as part of the project. They consist of each sensor board integrated onto a *MikroElektronika* 6LoWPAN clicker board [5]. This board runs a Real-Time Operating System (RTOS), which allows the board to respond very quickly to changes in sensor input, as well as the ability to be battery powered and the inclusion of a 6LoWPAN radio, to communicate with the base station.

Another requirement arising from the intended deployment scenario is that the system needs to be able to run on battery power, or indeed a low-voltage power source, for a considerable amount of time. The intention of the project is to reduce long-term study costs, and a need to replace sensor batteries regularly would be failing this. Consequently, the system would have to rely on hardware interrupts as much as possible, so that the devices can remain in a "sleep mode" when they are not needed.

1.2.1 Expected Outcomes And Deliverables

Below is a list of the expected outcomes and deliverables specified at the start of the project.

- A program built for the Creator Ci40 prototyping board that can read in an image sent by the external camera module, and classify the types and counts of animals in the image, before sending this data to a base station via a LORA connection
- A program for the IR clicker device to detect movement and send a message to a nearby camera device
- A program for the camera device to take a photo when it receives a command via 6LoWPAN and send it to the Ci40 board
- A simple cloud service to store and display results received from the prototype boards
- A rigorous testing regime for as much code as possible
- (If possible) a real-life test of the system in an uncontrolled environment (i.e. a park or zoo)

1.3 Research

The next step after gathering these requirements from the supervisor was to find existing research and solutions to similar problems. These papers, articles and reports have been compiled below.

1.3.1 Similar Projects, Papers and Articles

Towards Automatic Wild Animal Monitoring: Identification of Animal Species in Camera-trap Images using Very Deep Convolutional Neural Networks

In this paper by Gómez et al [14], a similar project is detailed to this one, in which the authors create a Convolutional Neural Network using the same Snapshot Serengeti [33] camera trap images. Their model uses "off-the-shelf" features of their chosen framework, and acheived a "top-5" accuracy of 88.9%, which is significant when compared to the crowdsourced classifications which obtained an accuracy of 96% [33].

Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning

This paper by Norouzzadeh et al [27] is in a similar tone to the paper published by Gómez et al [14]. They use the same Snapshot Serengeti dataset, but manage an accuracy of "over 93.8%", and can also "ìdentify count and describe animals" in the images (identifying features like whether the animals are standing or not, if they are moving or not, and so on). The authors conclude that they can "save 99.3% of the manual labour (over 17,000 hours) while performing at the same 96.6% accuracy level of human volunteers", and highlight the benefits of having more labour time for other scientific activities.

Where's The Bear?- Automating Wildlife Image Processing Using IoT and Edge Cloud Systems

The *Where's The Bear* project [10] involves a similar system to the one theorised in this project; however one key difference is that they run a server onsite, at a headquarters building, allowing them more easily transmit images from the camera traps to a server with higher processing power than a low-powered device. The authors crucially also deployed this to the wild with an accuacy of "at least 0.90". Another key difference is that rather than develop their own deep learning model, they used a technique called "transfer learning", to apply an existing model to a new problem domain (in this case, the Inception-v3 model [34]), which saved them a lot of time in having to design and train a model from the ground up.

Design and Implementation of an Open Source Camera Trap

This paper by Nazir et al [25] focuses on the deployment of remote, Internet-connected sensing devices in "the remotest part of the world". The focus of this project was to be able to have Internet access to the camera traps in a "harsh outdoor environment", using a satellite uplink for Internet connectivity, as well as being solar powered to increase the operating lifetime of the sensors as much as possible. Such a project like this demonstrates that the tools and technology exist to build an almost completely autonomous sensing system that can be deployed once and never has to be maintained. The authors also used more powerful hardware in their deployment than used

in this project, the Raspberry Pi B [11], which can easily handle more computationally-intensive tasks, like image processing and running deep learning models.

Chapter 2

Design

2.1 System Architecture

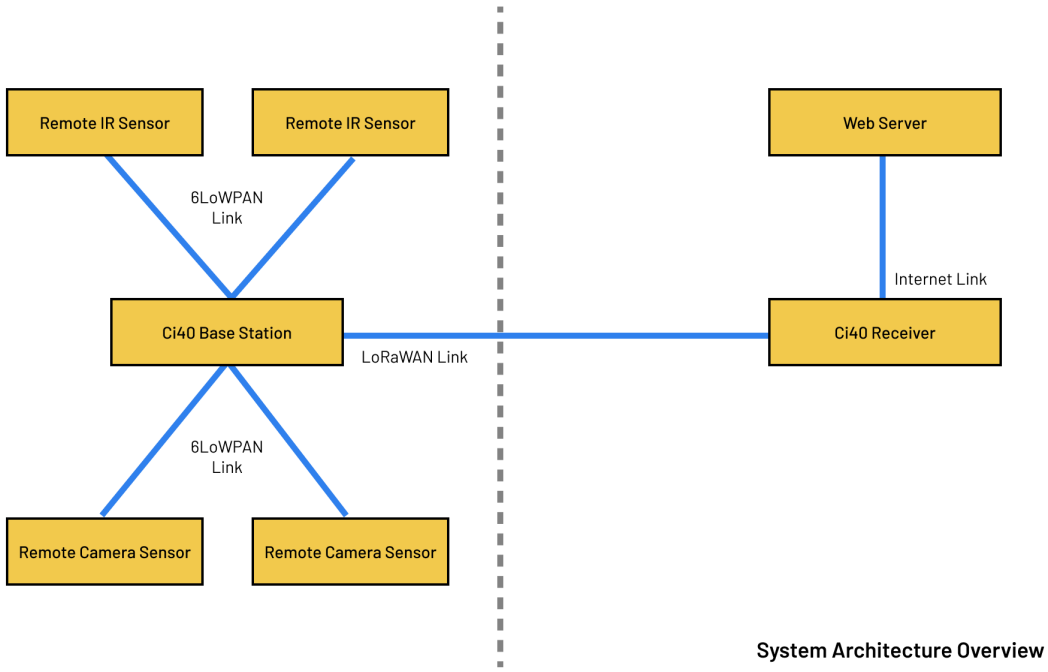

Figure 2.1: System architecture overview, showing how the high-level components of the system are connected.

Figure 2.1 shows the connections between the high-level components in the system, including the sensor deveices and the base stations, as well as the web server used to store and retrieve data.

2.1.1 Base Station

As previously mentioned in the requirements, the base station is a *Creator Ci40 IoT Hub* device [23]. It is a development board that is highly specialised for rapidly building Internet of

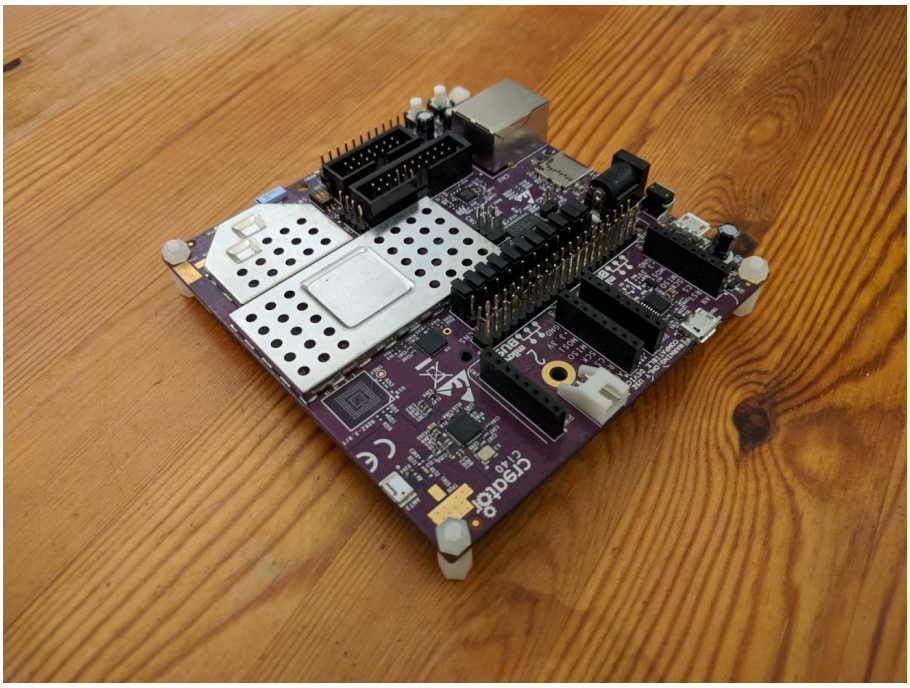

Figure 2.2: A photo of the Ci40 Base Station.

Things applications, due to its abundance of sockets, pins and radios for I/O.

Hardware Design Features

The Ci40 board includes a 6LoWPAN radio, which is crucial in allow it to conduct two-way communication with nearby sensing devices at a reasonably high data rate. The main drawback is the range, which is typically a couple of tens of meters [4]. However, this is perfect for communicating with the nearby sensor devices, which won't be too far away from the base station in the first place, and there is also the potential to use a mesh-style network, where communication might happen via one or more intermediary nodes. However, this would require sensor devices to be constantly active, thus preventing them from entering a lower power state and resulting in a much higher power usage.

Software Design

The base station provides a couple of very important tasks. Firstly, it acts as the main router, or coordinator, for nearby motion detector and camera pairs. It receives a motion alert from the motion sensor and sends a command to the related camera to capture an image. This program would also need to keep a list of sensor pairs, to ensure each command gets sent to the correct sensor.

In addition to this, the base station runs a program to process incoming images from the camera sensors and calculates the count and species of any wildlife in the photo. Since this is an image recognition problem, it was decided that the best solution would be to use a convoluted neural network, trained on a dataset of similar images, to estimate to a reasonable accuracy the

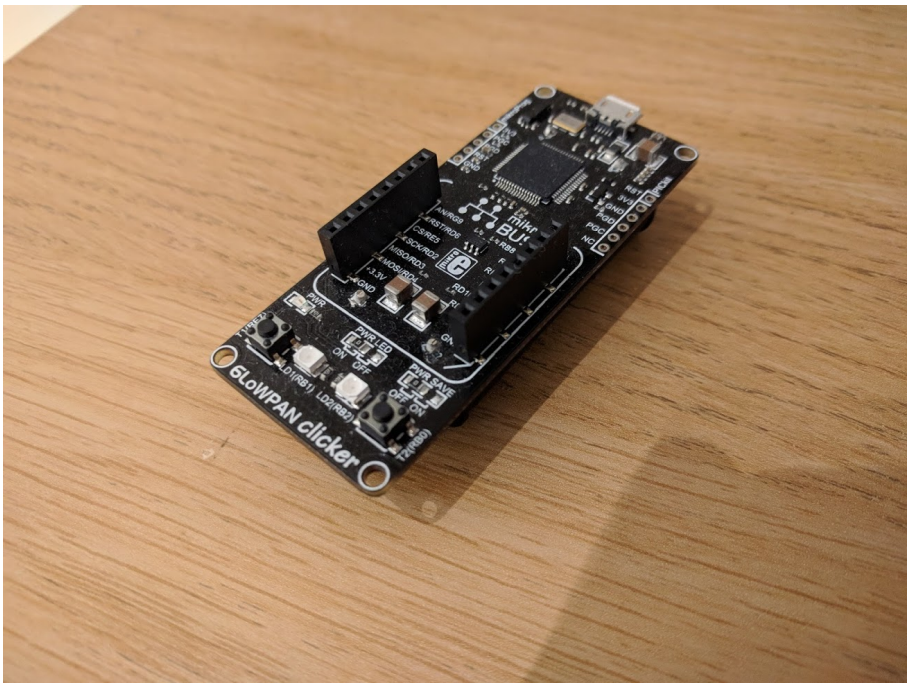

Figure 2.3: A photo of the *MikroElektronika* 6LoWPAN Clicker development board. Visible is the sixteen-pin mikroBUS interface, as well as a MicroUSB port to the rear.

species of wildlife in the image. However, since the Ci40 board uses a 32-bit processor architecture, it makes running neural networks a little more difficult, since most frameworks will only support 64-bit processors, as the larger word length results in larger computations being made possible.

Finally, the base station needs to act as a LoRaWAN transmitter, to send the count data to an Internet-connected base station, which can then in turn upload the results to the web server. So, the base station is running three programs at the same time to deal with different tasks. All of the code that deals with the 6LoWPAN connection or LoRaWAN connection is written in either C or Python, and uses the *LetMeCreate* library provided by the board manufacturer to increase ease of development.

2.1.2 Remote Sensors

The system also utilises a set of remote sensors that wirelessly connect with the base station using a 6LoWPAN connection. There's two kinds of sensors being used in the system, motion detector sensors and camera sensors. Both sensors use the same base board, a *MikroElektronika* 6LoWPAN Click Board [5], with either a camera or motion detector attachment.

Hardware Design

The *MikroElektronika* 6LoWPAN Click Board (see Figure 2.3), as previously mentioned, runs a Real-Time Operating System (RTOS) called Contiki. According to the Contiki project website [3], it is ideal for low-power IoT projects since it supports the full IPv4 and IPv6 networking suites, as well a being able to run on "tiny systems, only having a few kilobytes of memory available".

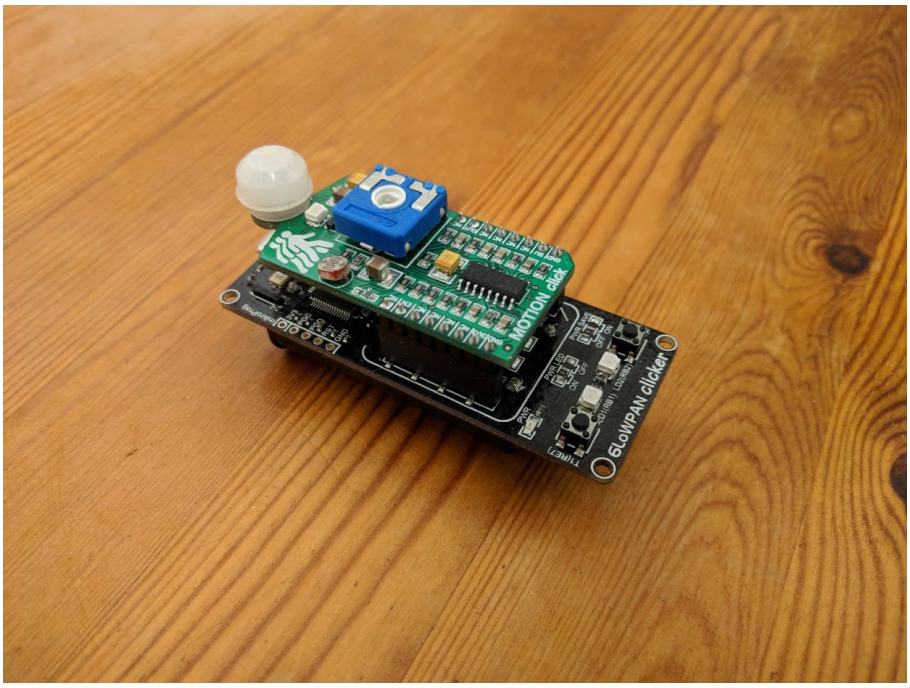

Figure 2.4: A photo of the motion sensor, consisting of the motion board installed on the 6LoWPAN clicker.

Creator, the manufacturers of the Ci40 base station, provide a toolchain for writing programs to the click board, as well as abstraction libraries for network and sensor interfaces [24].

Creator also provide a set of guides to help set up and write programs to the flash memory onboard the clicker device [22], which have proven invaluable. Code is written in a slightly modified version of the C programming language; the main difference being that all of the code runs in process threads that can be suspended, resumed and interrupted.

Communication

The remote sensors communicate with the base station using JSON-encoded strings sent via TCP over 6LoWPAN. A set of commands are defined and recognised by the sensors and the base station server alike, to allow messages to be sent that are both brief and human-readable, which is invaluable when debugging. The messages take a form similar to this:

```
{ "device_id": 1, "command": "heartbeat" }
```

A unique `device_id` is provided by every sensor to identify itself to the base station server, and on its first connection it will also provide a `pair_id` which tells the server which unique camera/motion detector pair the device belongs to. Each of these are provided to the sensor program at compile time, using environment variables.

Motion Detector Sensor

The motion detector sensor (see Figure 2.4) uses the 6LoWPAN Click board described above, with a *MikroElektronika* Motion Click device [8] integrated onto the board using the "mikroBUS"

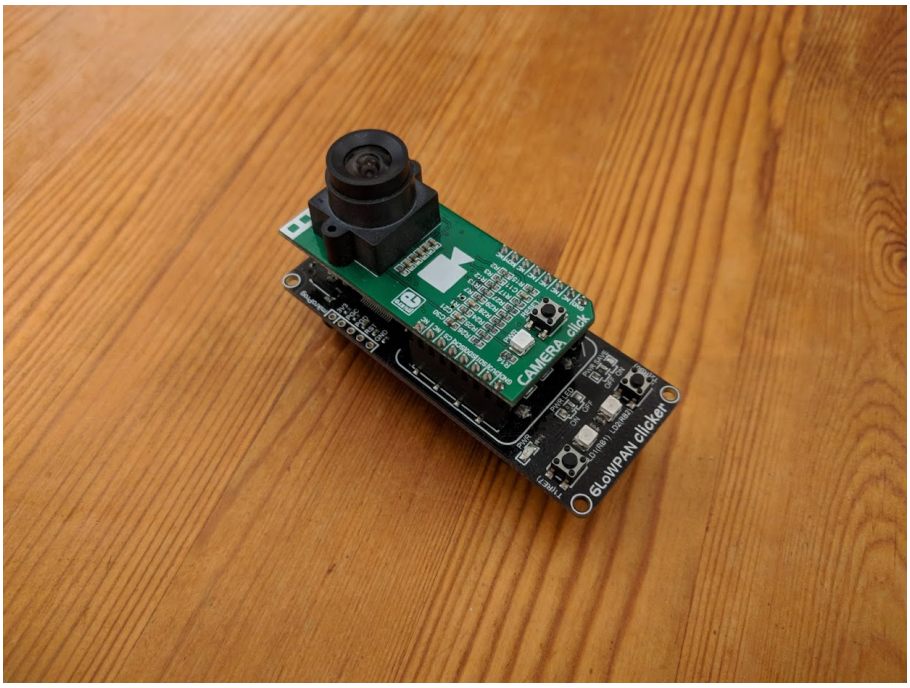

Figure 2.5: A photo of the camera sensor, consisting of the camera click board installed on the 6LoWPAN clicker.

port. According to the product page [8], it has a range of up to four metres which is probably not sufficient for real world usage. However, for prototyping purposes, it is perfect because of the ease of integration, thanks to the aforementioned *LetMeCreate* library [24].

The code runs in a continuous loop, that yields the main thread until it is resumed by an event interrupt. This event could be one of:

- a timer expiring,
- a TCP event (received packets, lost connection, et cetera),
- a motion detection event received from the motion sensor.

For the prototype version of this sensor, the code sends a "heartbeat" command to the base station every twenty seconds, for debugging purposes. But in a production version, the processor would only be be interrupted by TCP events or by motion detection events, as this would result in fewer interrupts over time and thus reduce the power usage of the device. The development version of the program also makes use of a debug server running on the base station. The clicker board does not always have a serial output available, so printing to console (i.e., using the `printf()` function) does not work. Therefore, Contiki includes a `PRINTF` macro that sends the string to a server using 6LoWPAN, if available.

Camera Sensor

The camera sensor (see figure 2.5) comprises of the same 6LoWPAN Clicker board, but instead of a motion sensor, there is a *MikroElektronika* Camera Click board [6] installed onto the mikroBUS

port. The board contains a digital camera sensor which, according to the specification page, has a maximum resolution of 640 by 480 pixels. It also contains an extra microcontroller, which "outputs the camera image to the target board microcontroller through the mikroBUS SPI interface". Essentially, it appears to transform the raw data stream from the camera into a data stream that can be sent using Serial Peripheral Interface bus (SPI) to the 'target' board (in this case, the 6LoWPAN clicker). However, a lot of the board's inner workings—save for the board schematics, available on the product page—is largely closed-source.

The code examples provided by *MikroElektronika* [7] helped to provide a little bit of insight into how the camera can be operated. For example, it provides a list of opcodes that the camera click accepts, such as requesting an image, or getting/setting a register on the camera sensor itself.

The main objective of the camera sensor is to respond to a message sent from the base station (over 6LoWPAN) commanding it to take a photo and send the photo back to the base station over the same connection.

2.1.3 Web Server

The web server serves the purpose of storing incoming species counts and types from any number of base stations, as well as keeping track of the locations of the base stations and sensors. Since the sensors and devices don't possess geolocation capabilities, this would be something that a research team using this solution would have to manually input.

As well as providing a way of uploading and storing this information, the web server would also have to provide methods of retrieving the data, as well as displaying the data. To this end, an API is the best solution for the problem. Appendix C shows a copy of the API specification that was created before development began on the API itself. Specifications are highly important, since they could be used to help develop comprehensive test suites for the code itself, as well as provided a solid foundation for any further documentation, for instance documentation that third parties use to build on top of the API.

The first stage in building the API is to model how the data is to be represented. This is achieved in three stages:

1. Work out what entities are to be represented with the API. For this project, this ended up being: the base stations, the associated sensor pairs, and the readings obtained from the sensor pairs. The web server doesn't need to concern itself with how the pairs connect to the base stations and to each other, so it is easier to model each motion sensor and camera sensor as a single pair.

2. Construct an entity-relationship diagram, to represent how each of the entities listed interact with each other. This also introduces the notion of multiplicity; for instance, how many base stations does a sensor pair interact with?

3. For each entity, deciding what data needs to be stored and accessible from the API. As mentioned earlier, a lot of data is actually excluded here since it is only relevant at a lower level. An example of data that is deliberately excluded is sensor IPv6 addresses, since they're only required by the base station for communication. This is also when it would be decided

what data is necessary and **must** be included for each instance of the entity, and what data is optional. Human-readable names are a good example of this.

From this initial requirements gathering, it is then possible to create an API specification, detailing how the API should react to certain input. It should be highly detailed, included specifying the HTTP response code that would be received under normal conditions.

The entire API specification is included in Appendix C for the reader's perusal.

Chapter 3

Implementation

It would be very difficult to attempt to build a complete working system during the course of the project, and that would be wholly out of scope. So throughout this project, the idea has been to build prototypes that demonstrate that a fully production-ready solution is viable.

A lot of the development work was also slowed down by a number of unforseen hardware issues, which are detailed in this chapter along with other implementation details.

3.1 Developing The Base Station Code

The main base station program is just a TCP server that handles commands coming in from the camera and motion sensors over the 6LoWPAN connection. Since the Linux kernel can already handle the 6LoWPAN connection, and no other hardware interfaces are required, there was a lot of flexibility in the choice of language and framework used to build the server. The Python language ended up being the choice of language, for its ease of development and pseudocode-like syntax.

The core language library also includes the `socket` library, which provides an easy to use, low-level interface for opening and accepting UDP and TCP connections. The server has a global dictionary that maps the id numbers of sensor pairs to the id of a camera sensor and the id of a motion sensor. This dictionary is populated from sensors which send an identification (`id`) command, broadcasting their unique `sensor_id` and their `pair_id`. This means that, when a motion sensor sends a motion detection command to the server, the server can look up the ID of the camera associated with the motion sensor and send it a command to capture a photo.

The source code for the base station server is available in section D.1 *(page 39)*.

3.1.1 Developing The Species Classifier

Less time was available in practice for developing the species classification software than intended, because of the complications arising from developing for the remote sensors as well trying to use the camera module. However, some exploration was made into the the techniques available to detect and classify species from a photo image.

Naturally, this was a problem well suited for a Convolutional Neural Network (CNN). CNNs, according to Christopher Olah's blog post on the subject [28], "can be thought of as a kind of

neural network that uses many identical copies of the same neuron". Each of these 'neurons' uses contains a mathematically trivial function, such as calculating mean averages of a set of pixel colours, to reduce the problem down. Weighted values are used to interpret these outputs and produce the final output of the network. These weights are "trained" (adjusted) using a training set of data, so that the network's output matches the expected output.

Because of the unforseen issues surrounding the 6LoWPAN connectivity and the camera sensor, there was less time than planned to develop a prototype classification program. However, some exploration was made into the techniques that could be used to build a classifier that can run on the Ci40 board.

Defining A Model

The first step to defining a deep learning model is to specify the input and the output. The input is a 172 by 144 pixel (also known as "QCIF format") image captured using the camera sensor, while the output is a probability distribution of size N, giving the probabilities that the content of the image is one of N different categories—in this case, it would be the list of animals we are classifying for, as well as a "no animal" category and a "unknown" category. And as for the intermediary steps in the model? That's where a trained Convolutional Neural Network is used.

CNNs are made up of a set of "layers", which are put together to form the network. According to the Stanford CS231n course notes [20], there are five main types of layers:

- Input layer—the raw data input into the network. For this project, the 172x144 pixel images from the camera sensor.
- Convolutional layer—a layer of neurons that each perform a compution on a small region, or a "kernel", of the image. Multiple "filters" may be used in this layer to detect different features. E.g. one filter may detect vertical edges, another may detect horizontal edges.
- Activation layer—applies some kind of activation function to each output from the previous layer to determine whether the neuron "fires" or not. The simplest example, also given by the CS231n course notes [20] is a threshold function like $max(0, x)$, which will have a range of $\{0, \mathbb{N}^+\}$.
- Pooling layer—Performs operations on groups of neurons to result in a smaller output. Useful for working on large, high-level features in images.
- Fully-connected layer—this is the final layer in the network, and takes *all* outputs from the previous layer and creates the final output; in this case it will produce a list of probabilities of the image being classified into a certain category/species.

Figure 3.1 shows a visualisation of how the layers of our neural network would interact with each other. Three-dimensional space is required since each layer is three-dimensional; the input, for example, takes an image with a width, height and three colour values for each pixel.

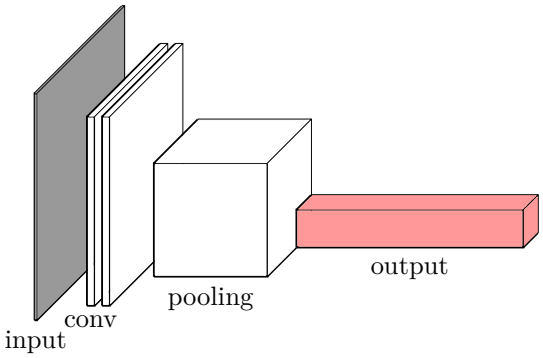

Figure 3.1: A visualisation of how the convolutional neural network would have been structured, with each image inputted as a square image with red, green, and blue colour channels; the one or more convolutional layers which filter for local features; a pooling layer which groups together low-level features to find high-level image patterns; and finally an output layer which ouputs a one-dimensional list of probabilities for each animal species.

Finding A Suitable Training Set

Manually building and labelling a set of images to train the model would have taken an extremely long time, since thousands of images would have to be obtained to help prevent the model from being overfitted to the training data (although other methods such as using the "Dropout" technique [31] can also reduce overfitting). For a production-ready system, a bespoke training set would be very suitable, especially for longer-term projects. However, for this prototype system, it was decided to use a pre-existing training set to focus purely on the deep learning model.

The first choice was to find a training set with woodland animals, which would be pertinent to this project. Unfortunately none were found, so the next best fit would be any kind of annotated camera trap data. Fortunately, a set of over four thousand camera annotated trap images, classified by "multiple experts", was published as part of the Snapshot Serengeti project [33] and released online [32]. The images are annotated with the number of species in the image, an id relating to one of 48 species in the dataset (or "unknown"), and the number of individual animals in the image.

Although this training data is not *exactly* the kind of data that would be collected in a production-ready version of this project, it provides a good enough picture of the kinds of challenges that may be faced by such a deep learning model in this project's application, such as views being obstructed by foliage, multiple species being visible, and images where it is impossible to determine what species is in the photo.

The script detailed in Section D.4 is used to download the Project Serengeti gold standard data set file (formatted as a CSV file), as well as the CSV file containing the URL of each image by ID. It then filters the list for the training set we want (images with only a single animal in the photo), fetches the actual photo, then resizes it into a common format (128x128 pixel JPEG file). This results in 1572 images to be used for testing and training. The test data is split, so that roughly the first eight percent (1250) of the images are used for training, and roughly the last twenty percent (322) are used for testing the model. A nice round number for the training set size is helpful so that the training can occur in batches.

Architectural Incompatibilities

The biggest drawback to using TensorFlow to develop an image classifier is that it only supports 64-bit architectures [36]. Therefore, it will not run as-is on a 32-bit processor, such as the one that the Ci40 uses. However, methods exist to generate a 32-bit model from a standard TensorFlow model exist, and are provided by the project itself. Arm, the mobile chip designer and manufacturer, provide a detailed tutorial for achieving this [21]. Theoretically it would then be possible to use this compiled 32-bit TensorFlow model onboard the Ci40 device for embedded classifier operations.

Another possible solution would have been to use an alternative deep learning framework. Other possibilities included using the Caffe deep learning framework [17], or a high-level framework like Keras [35]. However, the overwhelming amount of tutorials and documentation for TensorFlow available on the Internet meant that it was still a sensible choice of framework.

3.2 Developing The Remote Sensor Code

The 6LoWPAN clicker that the motion and camera sensors use runs on a Real-Time Operating System (RTOS) called *Contiki* [3]. The code that runs on these devices has to be flashed to the onboard flash memory. Therefore, Creator provide their own toolchain for compiling and flashing user code. This, however, was extremely difficult to set up, and a lot of time was spent obtaining the tooling, attempting to install the code, and being able to access the clicker from my computers. A lot of documentation was missing or not provided, which made independent investigations into the source code necessary.

Another issue was with writing the sensor code itself. The only provided documentation for Contiki is a handful of examples on its source code repository, as well as a tutorial on the Creator website [22]. To complicate matters further, the code used to program the boards is a modified version of the C language, except code runs in "process threads". However, after a lot of searching on the web, the Contiki wiki was discovered [9]. Despite being incredibly technical, there was helpful pieces of information available there to help decipher the inner workings of the Contiki platform, notably how the "protothreads" work.

Debugging the code was a further complication when developing the sensor code. The 6LoWPAN clicker only has a single MicroUSB port, which is used for flashing code to the onboard memory, and does not have a USB port of any kind to connect a serial terminal to. There is only two ways of debugging the clicker—sending text over the 6LoWPAN connection, or setting the two hardware LEDs on or off. A UDP-based debugging server is available along with a `PRINTF` macro, however these did not appear to work very well, if at all. It was also found that the debugging server would not work if the device was making a separate connection to the base station, such as when sending motion commands.

3.2.1 6LoWPAN issues

A reoccurring issue throughout the development of the remote sensors was the reliability of the 6LoWPAN connection. It often took multiple minutes or more for the remote sensors to connect to the base station, regardless of whether it was a TCP connection or a UDP connection. Sometimes,

the devices would not connect at all, and the debugging server (when it worked) reported multiple dropped messages. Since the Ci40 uses the 2.4 GHz frequency band, which is shared with many WiFi standards as well as Bluetooth, there is a possibility that this might be caused by interference between these devices, especially since all of the testing environments available during this project were in close proximity to WiFi links as well as Bluetooth devices, despite best efforts.

A manual published by *NXP Semiconductors* highlights the issues that can occur from the co-existence of these technologies on the same 2.4 GHz frequency band [37]. Notably, on page 19, they recommend that "to achieve satisfactory IEEE 802.15.4 [6LoWPAN] performance in the presence of WLAN interference, a channel centre-frequency offset of 7 MHz is recommended", and if 6LoWPAN is running on the same channel as the WiFi link, "a physical separation from the WLAN Access Point (AP) of 8 m is recommended". Essentially, they recommend either conducting radio operations away from WiFi AP, or selecting a different channel if possible. However, the WiFi APs in the testing environment are managed by a third party, so it was not possible to change their channels. Additionally, the documentation for the Ci40 base station was not clear enough on whether it was possible to set the 6LoWPAN wireless channel or not.

As a result of these 6LoWPAN issues, some modifications were made to the project. The camera sensor would be prototyped by installing the camera module *directly* onto the Ci40 base station, to make development and debugging easier. Since the Ci40 runs a full Linux operating system and full WiFi stack, it is possible to connect to it using an Secure Shell (SSH) connection. In addition to this, code designed to run on the camera clicker was not developed because it would be extremely similar to the motion sensor code, albeit with the camera processing code instead of the motion sensor event handling code.

3.2.2 Working With The Camera Click Module

One of the biggest obstacles during the project was working with the *MikroElektronika Camera Click* module [6], to be used as part of the prototype hardware for the remote camera sensor. The only documentation provided for this sensor is available on their "LibStock" website [7], but the only documentation provided is a board schematic and a code example, designed to be used with *MikroElektronika*'s own TFT display, with its own proprietary image format.

However, the code example provided was enough to discover roughly how the camera click board works and responds to input. It uses the SPI interface to receive commands and send a new or buffered image back to the parent device. The commands supported are, according to the code examples:

- `__CMD_WRITE_REG` (assumed) write to the registers on the camera Microcontroller Unit (MCU), to set things like exposure and brightness settings.
- `__CMD_READ_REG` (assumed) read from the registers on the camera MCU.
- `__CMD_GET_FRAME` capture a new photo and prepare to send it back through the SPI interface.
- `__CMD_GET_FRAME_BUFFERED` (assumed) get the currently buffered image from the camera, without capturing a new image.

- `__CMD_SET_ROW_NUM` unknown.
- `__CMD_SET_ROW_SIZE` unknown.

The SPI Interface

The SPI involves one "master" device, which sends commands to one or more "slave" devices. There is four data lines involved:

- CS—Chip Select. Selects the slave device to use.
- MISO—Transmits data from the master device to the slave device.
- MOSI—Transmits data from the slave device to the master device.
- CLK—Timing signal.

The master and slave devices transmit data as if they were one "rotary" system. Each side has a buffer of a fixed size (in the case of the camera module, 8 bits), and as the master device pushes a bit to the SPI interface, the slave device sends a bit back. This continues until the buffers are essentially swapped around (see figure 3.2). One peculiarity in this implementation of SPI is that the camera module use sa separate "interrupt" pin (provided by the mikroBUS standard) to let the master device know that it is ready for commands.

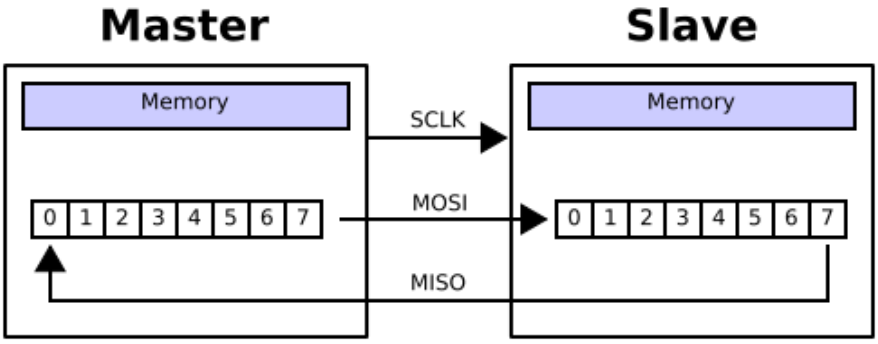

Figure 3.2: A diagram showing data being swapped between two SPI devices. Image created by Cburnett on Wikimedia Commons. Obtained from Wikimedia Commons under the GNU Free Documentation License. Source: `https://commons.wikimedia.org/wiki/File:SPI_8-bit_circular_transfer.svg`

Getting Camera Output

The code in Section D.2 shows the code used to attempt to obtain output from the camera sensor, installed onto one of the Ci40's mikroBUS ports. It works by initialising the SPI pins and setting speed and other settings, before waiting for the camera to send a "ready" signal and then sending the SPI command to get an image from the camera. It then reads the incoming data to a buffer, and saves that buffer to a file. The data *should*, in theory, be the image data from the camera. This code should work since it roughly follows the steps used in *MikroElektronika*'s example code.

However, the camera module frequently returns a lot of blank data. Below is one example of data received from the camera sensor after sending a `__CMD_GET_FRAME` command:

```
[mbell@chancery-lane ~]$ hexdump ~/out.bin
0000000 ffff ffff ffff ffff ffff ffff ffff ffff
*
000c600 0000 0000 0000 0000 c611
000c60a
```

Interpreting Camera Output

Sometimes the camera would send data that was not just a stream of `1` bits, however it was not clear whether the returned data was nonsense, or if the data was in some proprietary data format.

Figure 3.3 shows the data previewed as a *YUV* image. This is an 8-bit pixel encoded image format that contains three values for each colour—the brightness (also called luminance), and two values defining the colour itself (the chroma) [38]. The data was estimated to be using some kinda of YUV format, because of reoccurring patterns in the data that suggested that pixel data was set as some 4-bit value, then two 2-bit values.

The possibility of the data being stored as Red, Green, Blue (RGB) data was ruled out when attempting to parse the raw image file as such. The results can be seen in figure 3.4. It is clear that the data is not RGB since the rendering software runs out of bits roughly two-thirds of the way through parsing the image. Since public access to the source code for the camera module was prohibited by the manufacturer, it was impossible to tell whether the image format chosen was incorrect, or if the camera module itself was defective or damaged. Therefore, no image was ever able to be correctly obtained from the camera module.

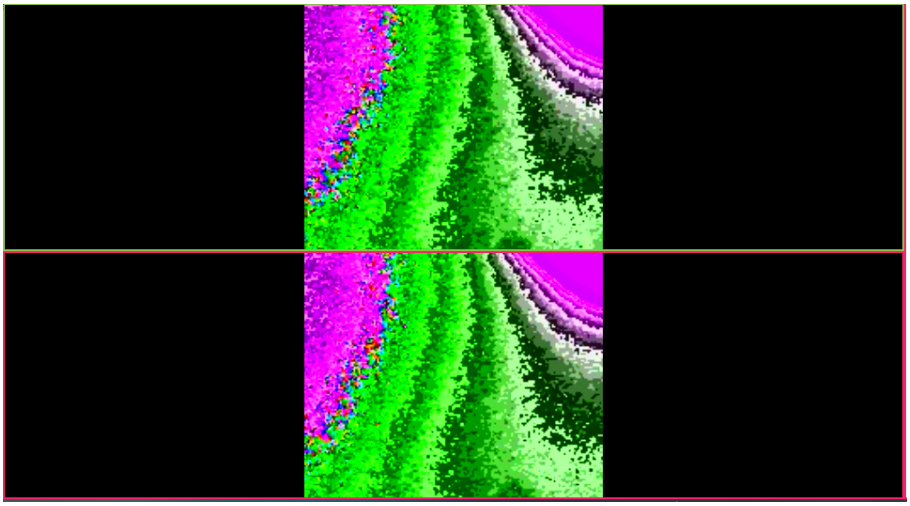

Figure 3.3: The data received from the camera sensor, interpreted as a YUV-formatted image.

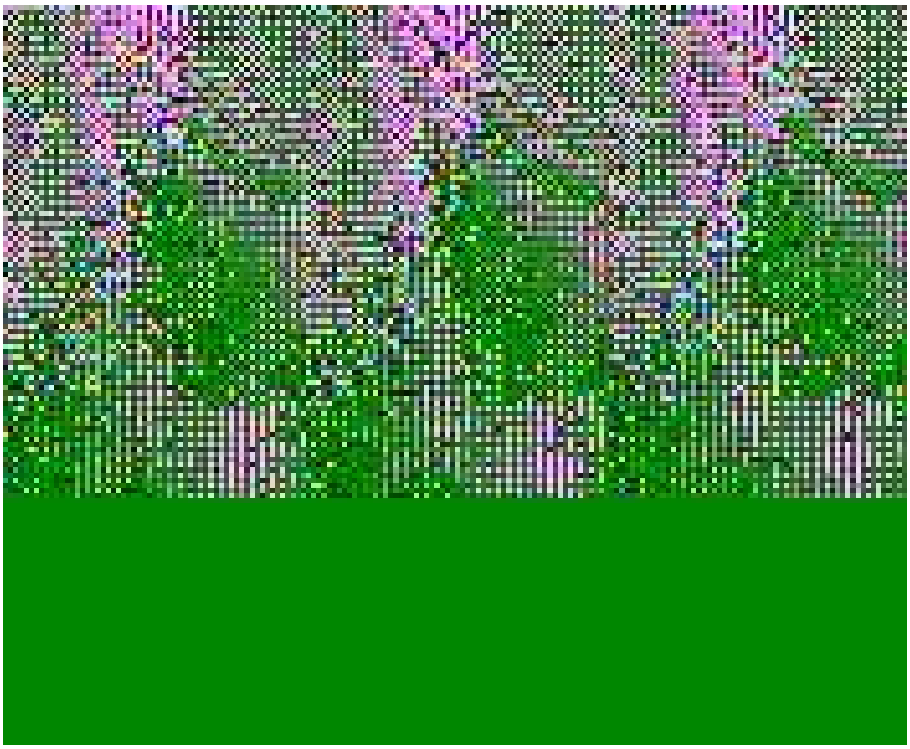

Figure 3.4: The data received from the camera sensor, interpreted as a RGB-formatted raw image.

Read-Only Register Random Number Generator

The camera module used on the camera sensor is the OV7670-VL2A CMOS sensor, and a datasheet is readily available online [29]. Included in the datasheet is a list of the sensor's registers, detailing each register's:

- One-byte address (in hexadecimal),
- Name of the register,
- The default hexadecimal value,
- Whether it can be read and/or written to,
- And a description of what the register represents.

Two of these registers were of significance for testing the SPI interface, addresses `1C` and `1D`. These are both read-only registers that return the high byte and low byte (respectively) of the two byte manufacturer ID. They are both read only and are supposed to always return `0x7F` and `0xA2` respectively.

However, when this was attempted on the `camera_base` application (see Section D.2 lines 141–144), the program received random numbers each time (see Figure 3.5 for server output). Either one of two explanations is possible—either the code to read from the register is incorrect, or the camera click board is defective. As with trying to read the image itself, it is impossible to conclude which of these explanations is correct without being able to access the *MikroElektronika* source code.

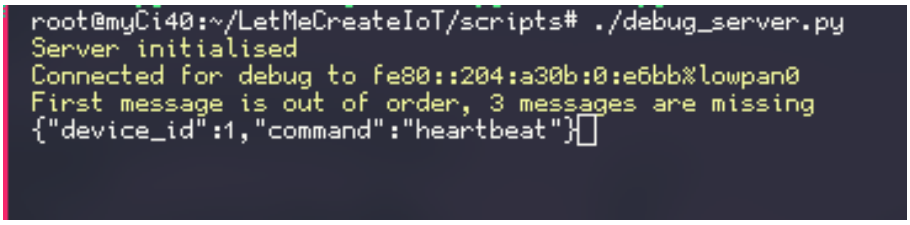

Figure 3.5: Screenshot of a terminal image showing the debugging server missing messages from the remote sensor.

3.3 Developing The API Server

The API server was built entirely using the Node.js framework [26], which runs JavaScript code on computers and servers without a web browser. It was chosen for ease of development, and since the server would be managing a JSON API, it made sense to use a language which natively supports JSON too.

The server framework used was Koa [19], which supports modern JavaScript features such as asynchronous functions. Almost all helper functionality (such as URL routing) is not provided as part of the core package, meaning that the library has a very small blueprint—this is very useful for fast deployment to cloud servers. The database of choice was PostgreSQL; this again was a choice made because of similarity with the program, as well as very few differences between it and its competition (for example, MySQL and MariaDB).

All of the code was developed using a Behaviour-Driven Development (BDD) methodology, meaning that tests were written before the code itself. This is expanded on in more detail in the *Testing* chapter. The final code can be found in the appendices.

Chapter 4

Testing

4.1 API Server Testing

To ensure that the API server works with as few errors and incorrect responses as possible, a series of automated test suites were set up that could be run on the codebase. Two different types of test were used—functional (unit) tests, that run the actual models and methods to ensure they act as expected, and integration tests, that perform actual HTTP requests to the server to check that it returns the expected output, and that the output is formatted correctly.

4.1.1 Testing Framework

The Jest [16] framework, an open source testing framework maintained by Facebook, was the natural choice of test framework for the project, thanks to its ease of setup and use, as well as a very straightforward library. Jest exposes an `expect` object that allows the developer to write Behaviour-Driven Development (BDD)-style tests that read almost like sentences, such as "expect this status code to be 200". The Jest project also feature very extensive documentation about using the `expect` object on their website [15].

To ensure strict code style and readability, a code linter was also integrated into the project. The chosen linter was `ESLint`, a popular JavaScript linter that can highlight formatting issues, inaccessible ("dead") code, and other style violations that may lead to unintentional behaviour. One example of this is comparing two variables with the *equality* operator (`==`) rather than using the *strict equality* operator (`===`), which additionally tests for type equality as well as value equality.

Formatting rules specified in the Airbnb JavaScript Style Guide [1] were used as the main rules for the linter, as well as rules introduced by the `Prettier` formatter, which formats the code when the user saves. The Airbnb JavaScript Style Guide is a popular code style for JavaScript—as of 2018-03-23, nearly seventy thousand users have "starred" the GitHub repository [1]. This means that it is a rather standard way of formatting code, and should help to increase the chance of any possible collaboration in the future.

A test script was also added to the root of the project, inside the `package.json` file, that allows for all of the project's tests to be run at once. This was then integrated with the JavaScript

packages `husky` and `lint-staged` so that the code was linted, formatted, and all the tests run prior to a commit being created. The commit would then be cancelled if there was a linter error or one or more tests failed.

4.1.2 Creating The Unit Tests

The unit tests were derived from the expectations held on how the code representations of the data models (for instance, the JavaScript classes that represent the base stations, sensor pairs and readings) are initialised with correct values, and validate their possible input values correctly. For instance, each sensor pair has a `camera_id` and a `motion_id` that stores the identifiers for the camera and motion sensors in that pair. Those IDs should be positive integers, so tests must be devised that ensure that the model accepts positive integers and rejects any other input, by testing the model against as many different inputs as possible.

4.1.3 Creating The Integration Tests

The integration tests involve testing that the server as a whole works as intended, that when an end user sends correctly-formatted input they get the expected output, and if they don't they get the correct error message. This is sometimes referred to as textitblack-box testing, since you are not testing the code itself—rather, you are testing the *system* as if you are the end user.

To help set up and run the HTTP requests and test the server responses, the `supertest` library [13] was used. However, one component of the system that isn't present during testing is the database. Fortunately, a library exists [12] that can "mock" (replicate the functionality) of the database connection library so that response to database queries coming from the server can be crafted to match various scenarios, such as no matching data existing on the database, or mocking the data being written to the database.

4.1.4 Test Results And Coverage Report

In the submitted API code, all forty-six of the tests written pass successfully, and none are failing, which is what is expected from code that was written using the Test-Driven Development (TDD) methodology (write the tests, then write the code to pass the tests). This also reflects how code would be written in industry to meet quality control standards. However, the test pass rate is only one half of the story, because it counts for nothing if the tests don't cover enough of the application's code. Jest provides a test coverage analyser as part of its test suite, which analyses the project's code and returns a percentage of how much of the code is covered by tests. Table 4.1.4 shows the output from the Jest test coverage report. Overall, the tests cover 82.9% percent of all code statements made, 60.7% of all possible code branches, 73.1% of all functions, and 82.7% of all lines.

This statistic is reasonably high for a small project, but to improve reliability of the server, more tests can and should be added in the future. Ideally, one should always aim to achieve 100% test coverage, but time constraints can impair this. Zhu et al discuss the various merits of these test coverage metrics in great detail in *Software Unit Test Coverage and Adequacy* [39], but

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	**82.91**	**60.71**	**73.17**	**82.74**	
srv	76.3	52.27	55	76.3	
baseStation.js	94.64	66.67	100	94.64	46,71,108
index.js	89.19	61.11	60	89.19	30,58,62,63
knexfile.js	100	100	100	100	
sensorPair.js	39.02	0	0	39.02	... 69,70,71,73,77
srv/models	96.55	91.67	90.48	96.43	
BaseStation.js	100	100	100	100	
Reading.js	96.43	83.33	88.89	96.15	115
SensorPair.js	93.75	100	83.33	93.75	57
index.js	100	100	100	100	
srv/test	100	100	100	100	
db.js	100	100	100	100	

Table 4.1: Jest test coverage report, generated on Monday 23rd April 2018.

combined these metrics serve as a useful insight into how effectively the tests actually test the code written.

All of the server tests can be found in the source code. Any file named `*.test.js` will contain tests.

Chapter 5

Evaluation and Conclusion

5.1 Evaluation of Expected Deliverables

Below is a list of the expected outcomes and deliverables specified at the start of the project and if they have been achieved (and to what extent).

5.1.1 Base Station Program

"A program built for the Creator Ci40 prototyping board that can read in an image sent by the external camera module, and classify the types and counts of animals in the image, before sending this data to a base station via a LORA connection."
This deliverable was only partially completed. A small program was created that can handle sensors connecting to the base station, as well as receiving motion events, but the classification program was not built.

5.1.2 Motion Sensor Program

"A program for the IR clicker device to detect movement and send a message to a nearby camera device."
This deliverable was completed and works to a reasonable standard, despite issues with the onboard wireless radios.

5.1.3 Camera Sensor Program

"A program for the camera device to take a photo when it receives a command via 6LoWPAN and send it to the Ci40 board."
This deliverable was not completed, because of multiple issues using the provided camera sensor and getting any readable data from it.

5.1.4 Cloud Server

"A simple cloud service to store and display results received from the prototype boards".

This deliverable was completed to a high standard.

5.1.5 Tests

"A rigorous testing regime for as much code as possible."
This deliverable was met for the cloud server, as well as acceptance testing being conducted for the motion sensor.

5.1.6 Real-Life Test

"(If possible) a real-life test of the system in an uncontrolled environment (i.e. a park or zoo)."
This was not met due to the other incomplete deliverables.

5.2 Evaluation of Achievements

Despite not all goals being met, the project should overall be considered successful. The research conducted early in the project highlighted a clear need for such an autonomous system to be developed, and recent strides in the field of computer vision mean that such systems are becoming faster and cheaper to deploy. However, it was a shame that not all of the deliverables could be completed, especially being able to test the system in a real-life setting.

One of the biggest and persistent issues throughout the project was problems developing on the given hardware and using it to the best of its ability. These problems were discussed in the *Implementation* chapter, including lack of documentation regarding the libraries used to connect to sensor and hardware, as well as wireless connections not working as intended. In hindsight, perhaps it would have been best to swap the hardware out for better documented, more reliable (although perhaps a lot less realistic) hardware, like the Arduino platform [2]. The devices may not be as energy efficient as the ones used in this project, but that could be considered a less important characteristic for a technical demonstration.

Also, it may have been wiser to begin researching and developing the deep learning classifier earlier on in the project. The work plan, devised at the start of the project, mentioned researching and developing this section of the project from November onwards, whereas perhaps research into different techniques and frameworks should have been made earlier on.

5.3 Possible Future Work

There is a lot of promise in developing deep learning solutions to such a problem as automatically classifing animals in camera trap images, and a lot of recent breakthroughs have been extremely exciting—including the work of Elias et al [10] whose work was published since the start of this project. Perhaps the rise of general yet powerful models for computer vision such as Inception-v3 [34] remove most of the need for bespoke models to be created to solve these kinds of problems, allowing more and more people to build exciting and powerful applications for computer vision classification.

Additionally, leaps in the field of mobile computing are also incredibly exciting. New development boards are appearing every day that can run on very small amounts of power, vastly increasing expected lifetimes of such projects as this one, as well as reducing costs needed to maintain and replace sensors; perhaps future work in this domain could include building clustered networks of sensors that can be highly resilient to link disruption, yet still consume just a small amount of power.

Appendix A

Glossaries

Glossary

Symbols

6LoWPAN Short-range wireless data transmission standard. Short for "IPv6 over LOw Power Wireless Personal Area Networks"; alternative to protocols like Bluetooth and Zigbee. 3, 7–10, 13, 14, 16, 17

C

Contiki A Real-Time Operating System (RTOS) designed specifically for the Internet of Things. Contains a full network stack and can run on a minimal system, with lower power consumption. 8

L

LoRaWAN Wireless data transmission standard designed for long range communication at low power, at the cost of a lower data transmission rate. 2, 8

M

mikroBUS Bus standard for integrating IoT sensors onto development boards. Contains pins for SPI, analog data transmission, power, and an interrupt pin. 8, 10, 18, 19

MIPS Multiprocessor without Interlocked Pipeline Stages, a type of processor architecture. 3

O

opcode Short for operation code, a command or instruction that may be part of a device's instruction list. 10

overfit Where a statistical or machine learning model is adjusted so that it works extremely well on training data, but does not work so well on non-training data.. 15

Acronyms

A

AP Access Point. 17

API Application Programming Interface. 2, 11, 12, 20, 22, 23

B

BDD Behaviour-Driven Development. 20, 22

C

CNN Convolutional Neural Network. 4, 13, 14

CSV Comma-Separated Values file. 15

G

GHz Gigahertz. 17

H

HTTP Hyper-Text Transfer Protocol. 12, 22, 23

I

I/O Input/output. 7

IoT Internet of Things. 1, 6–8

IPv4 Internet Protocol version 4. 8

IPv6 Internet Protocol version 6. 8, 12

J

JSON JavaScript Object Notation. 9, 20

M

MCU Microcontroller Unit. 17, 18

R

RGB Red, Green, Blue. 19

RTOS Real-Time Operating System. 3, 8, 16

S

SPI Serial Peripheral Interface bus. 10, 17–20

SSH Secure Shell. 17

T

TCP Transmission Control Protocol. 9, 10, 13, 17

TDD Test-Driven Development. 23

U

UDP User Data Protocol. 16, 17

Appendix B

Bibliography

[1] Airbnb. *Airbnb JavaScript Style Guide()*. URL: https://github.com/airbnb/javascript/ (visited on 04/23/2018).

[2] *Arduino Home*. URL: https://www.arduino.cc/ (visited on 04/29/2018).

[3] Contiki. *Contiki: The Open Source Operating System for the Internet of Things*. URL: http://www.contiki-os.org (visited on 04/13/2018).

[4] David Culler and Samita Chakrabarti. "6LoWPAN: Incorporating IEEE 802.15. 4 into the IP architecture". In: *IPSO Alliance, White paper* (2009).

[5] MikroElektronika d.o.o. *6LoWPAN clicker - a compact development board | MikroElektronika*. URL: https://www.mikroe.com/clicker-6lowpan (visited on 04/12/2018).

[6] MikroElektronika d.o.o. *Camera click — board with OV7670-VL2A CMOS image sensor*. URL: https://www.mikroe.com/camera-click (visited on 04/15/2018).

[7] MikroElektronika d.o.o. *LibStock - Camera Click - Example*. URL: https://libstock.mikroe.com/projects/view/1263/camera-click-example (visited on 04/15/2018).

[8] MikroElektronika d.o.o. *Motion click - pir500b motion detector sensitive only to live bodies*. URL: https://www.mikroe.com/motion-click (visited on 04/13/2018).

[9] Contiki Developers. *Home, contiki-os/contiki Wiki*. URL: https://github.com/contiki-os/contiki/wiki (visited on 04/25/2018).

[10] Andy Rosales Elias et al. "Where's the Bear?-Automating Wildlife Image Processing Using IoT and Edge Cloud Systems". In: *Internet-of-Things Design and Implementation (IoTDI), 2017 IEEE/ACM Second International Conference on*. IEEE. 2017, pp. 247–258.

[11] Raspberry Pi Foundation. *Raspberry Pi 1 Model B+ - Raspberry Pi*. URL: https://www.raspberrypi.org/products/raspberry-pi-1-model-b-plus/ (visited on 04/28/2018).

[12] colonyamerican on GitHub. *colonyamerican/mock-knex: A mock knex adapter for simulating a database during testing*. URL: https://github.com/colonyamerican/mock-knex/ (visited on 04/23/2018).

[13] visionmedia on GitHub. *visionmedia/supertest: Super-agent driven library for testing node.js HTTP servers using a fluent API.* URL: https://github.com/visionmedia/supertest (visited on 04/23/2018).

[14] Alexander Gomez, Augusto Salazar, and Francisco Vargas. "Towards automatic wild animal monitoring: identification of animal species in camera-trap images using very deep convolutional neural networks". In: *arXiv preprint arXiv:1603.06169* (2016).

[15] Facebook Inc. *Expect · Jest.* URL: https://facebook.github.io/jest/docs/en/expect.html (visited on 04/23/2018).

[16] Facebook Inc. *Jest • Delightful JavaScript Testing.* URL: https://facebook.github.io/jest/ (visited on 04/23/2018).

[17] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[18] K Ullas Karanth. "Estimating tiger Panthera tigris populations from camera-trap data using capture—recapture models". In: *Biological conservation* 71.3 (1995), pp. 333–338.

[19] *Koa - next generation web framework for node.js.* URL: http://koajs.com/ (visited on 04/29/2018).

[20] Stanford University Vision Lab. *CS231n Convolutional Neural Networks for Visual Recognition.* URL: http://cs231n.github.io/convolutional-networks/#layers (visited on 04/29/2018).

[21] Arm Limited. *Machine Learning on Arm | Generate an optimized 32-bit model – Arm Developer.* URL: https://developer.arm.com/technologies/machine-learning-on-arm/developer-material/how-to-guides/optimizing-neural-networks-for-mobile-and-embedded-devices-with-tensorflow/generate-an-optimized-32-bit-model (visited on 04/29/2018).

[22] Imagination Technologies Limited. URL: https://docs.creatordev.io/clicker/guides/quick-start-guide/ (visited on 04/13/2018).

[23] Imagination Technologies Limited. *Creator Ci40 IoT Hub - Build your Internet of Things Product.* URL: https://creatordev.io/ci40-iot-hub.html (visited on 04/12/2018).

[24] Imagination Technologies Limited. *CreatorDev/LetMeCreate.* URL: https://github.com/CreatorDev/LetMeCreate (visited on 04/13/2018).

[25] Sajid Nazir et al. "Design and implementation of an open access camera trap". In: *Digital Conservation Conference*. 2014.

[26] *Node.js.* URL: http://nodejs.org (visited on 04/29/2018).

[27] Mohammed Sadegh Norouzzadeh et al. "Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning". In: *arXiv preprint arXiv:1703.05830v5* (2017).

[28] Christopher Olah. "Conv nets: A modular perspective". In: *URL http://colah. github. io/posts/2014-07-Conv-Nets-Modular* (2014).

[29] *OV7670/OV7171 CMOS VGA(60x480) CameraChip™ with OmniPixel® Technology*. Version 1.01. OmniVision Technologies. July 8, 2005.

[30] Leandro Silveira, Anah TA Jacomo, and Jose Alexandre F Diniz-Filho. "Camera trap, line transect census and track surveys: a comparative evaluation". In: *Biological Conservation* 114.3 (2003), pp. 351–355.

[31] Nitish Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[32] Alexandra Swanson et al. *Data from: Snapshot Serengeti, high-frequency annotated camera trap images of 40 mammalian species in an African savanna*. URL: https://doi.org/10.5061/dryad.5pt92 (visited on 04/29/2018).

[33] Alexandra Swanson et al. "Snapshot Serengeti, high-frequency annotated camera trap images of 40 mammalian species in an African savanna". In: *Scientific data* 2 (2015), p. 150026.

[34] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826.

[35] Keras Team. *Keras Documentation*. URL: https://keras.io/ (visited on 04/29/2018).

[36] Tensorflow. *Installing TensorFlow | TensorFlow*. URL: https://www.tensorflow.org/install/ (visited on 04/29/2018).

[37] NXP Laboratories UK. *Co-existence of IEEE 802.15.4 at 2.4 GHz Application Note*. Tech. rep. Nov. 8, 2013.

[38] Christopher Wright. *YUV Colorspace*. URL: http://softpixel.com/~cwright/programming/colorspace/yuv/ (visited on 04/27/2018).

[39] Hong Zhu, Patrick AV Hall, and John HR May. "Software unit test coverage and adequacy". In: *Acm computing surveys (csur)* 29.4 (1997), pp. 366–427.

Appendix C

API Specification

C.1 Base Station

Endpoint: `/basestation/:id`
This endpoint concerns the base stations registered on the system.

Optional Parameters

- `:id` —The id of the selected base station (not required for getting list of all base stations or adding new station).

C.1.1 Get a list of all base stations

Method: `GET`
Request Format: `<n/a>`
Response Content-Type: `JSON`
Expected Status Code: `200 OK`
Example Response:

```
{
  "error": "",
  "content": [
    {
      "id": "00000000-0000-0000-0000-000000000000",
      "name": "Test Station",
      "lat": 0,
      "lng": 0
    }
  ]
}
```

C.1.2 Add a new base station

Method: `POST`
Request Format: `JSON`

```
{
  "name": "Test Station",
  "lat": 0,
  "lng": 0,
}
```

Required Fields: <none>
Response Content-Type: JSON
Expected Status Code: 201 Created
Example Response:

```
{
  "error": "",
  "content": {
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "Test Station",
    "lat": 0,
    "lng": 0
  }
}
```

C.1.3 Modify an existing base station

Method: PATCH
Request Format: JSON

```
{
  "name": "Test Station",
  "lat": 0,
  "lng": 0,
}
```

Required Fields: <none>
Response Content-Type: JSON
Expected Status Code: 200 OK
Example Response:

```
{
  "error": "",
  "content": {
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "Test Station",
    "lat": 0,
    "lng": 0
  }
}
```

C.1.4 Delete an existing base station

Method: DELETE
Request Format: <n/a>
Required Fields: <none>

Response Content-Type: `<none>`
Expected Status Code: `204 No Content`

C.2 Base Station Sensor Pairs

Endpoint: `/basestation/:id/sensorpairs`
This endpoint concerns the sensor pairs connected to a base station.

C.2.1 Get a list of a base station's sensor pairs

Method: `GET`
Request Format: `<n/a>`
Response Content-Type: `JSON`
Expected Status Code: `200 OK`
Example Response:

```
{
  "error": "",
  "content": [
    {
      "id": "00000000-0000-0000-0000-000000000000",
      "name": "Treetop Sensor",
      "lat": 0,
      "lng": 0,
      "camera_id": 234,
      "motion_id": 120
    }
  ]
}
```

C.2.2 Add a new sensor pair to a base station

Method: `POST`
Request Format: `JSON`

```
{
  "name": "Treetop Sensor",
  "lat": 0,
  "lng": 0,
  "camera_id": 123,
  "motion_id": 456
}
```

Required Fields: `camera_id, motion_id`
Response Content-Type: `JSON`
Expected Status Code: `201 Created`
Example Response:

```
{
  "error": "",
  "content": {
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "Treetop Sensor",
    "lat": 0,
    "lng": 0,
    "camera_id": 123,
    "motion_id": 456
  }
}
```

C.3 Sensor Pair

Endpoint: `/sensorpair/:id`
This endpoint concerns all sensor pairs registered on the system.

Optional Parameters

- `:id` —The id of the selected sensor pair (not required for getting list of all sensor pairs or adding new pair).

C.3.1 Get a list of all sensor pairs

Method: `GET`
Request Format: `<n/a>`
Response Content-Type: `JSON`
Expected Status Code: `200 OK`
Example Response:

```
{
  "error": "",
  "content": [
    {
      "id": "00000000-0000-0000-0000-000000000000",
      "name": "Treetop Sensor",
      "lat": 0,
      "lng": 0,
      "camera_id": 123,
      "motion_id": 456
    }
  ]
}
```

C.3.2 Modify an existing sensor pair

Method: `PATCH`
Request Format: `JSON`

```
{
  "name": "Treetop Sensor",
  "lat": 0,
  "lng": 0,
}
```

Required Fields: <none>
Response Content-Type: JSON
Expected Status Code: 200 OK
Example Response:

```
{
  "error": "",
  "content": {
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "Treetop Sensor",
    "lat": 0,
    "lng": 0,
    "camera_id": 123,
    "motion_id": 456
  }
}
```

C.3.3 Delete an existing sensor pair

Method: DELETE
Request Format: <n/a>
Required Fields: <none>
Response Content-Type: <none>
Expected Status Code: 204 No Content

C.4 Sensor Pair Reading

Endpoint: /sensorpair/:pairid/reading/:readingid
This endpoint concerns the base stations registered on the system.

Required Parameters

- :pairid —The id of the selected sensor pair.

Optional Parameters

- :readingid —The id of the selected reading (not required for getting list of all readings or adding new reading).

C.4.1 Get a list of all readings for a sensor

Method: GET

Request Format: <n/a>
Response Content-Type: JSON
Expected Status Code: 200 OK
Example Response:

```
{
  "error": "",
  "content": [
    {
      "id": "00000000-0000-0000-0000-000000000000",
      "t": "2018-01-01T12:00:00+00:00",
      "counts": [
        {
          "species_id": 12,
          "name": "Blackbird",
          "count": 1
        }
      ]
    }
  ]
}
```

C.4.2 Add one or more readings to a sensor pair

Method: POST
Request Format: JSON

```
[
  {
    "t": "2018-01-01T12:00:00+00:00",
    "counts": [
      {
        "species_id": 12,
        "count": 1
      }
    ]
  }
]
```

Required Fields: t, counts
Response Content-Type: JSON
Expected Status Code: 201 Created
Example Response:

```
{
  "error": "",
  "content": [
    {
      "id": "00000000-0000-0000-0000-000000000000",
      "t": "2018-01-01T12:00:00+00:00",
      "counts": [
        {
          "species_id": 12,
```

```
            "name": "Blackbird",
            "count": 1
          }
        ]
      }
    ]
  }
```

Appendix D

Code Listing

D.1 base/__main__.py

```
#
# A very simple TCP server.
# Just listens to port 9876 and prints to STDOUT
# Plays nicely with `tee` as a result.
import signal
import socket
import sys
from datetime import datetime
import json

listen_sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
listen_sock.bind(("::", 9876))

device_pairs = {}

# Maps device ids to their pair ids
device_ids_to_pairs = {}


def command_id(obj, conn):
    try:
        pair_id = obj["pair_id"]
        device_id = obj["device_id"]
        device_type = obj["type"]
    except KeyError:
        print("missing keys. make sure `pair_id`, `device_id` and `type` are provided.")
        return
    print("device {} to be paired in pair {}".format(device_id, pair_id))
    if not pair_id in device_pairs:
        device_pairs[pair_id] = {}
    if not device_id in device_ids_to_pairs:
        device_ids_to_pairs[device_id] = device_id
    device_pairs[pair_id][device_type] = device_id
    print("device pair {}: now {}".format(pair_id, device_pairs[pair_id]))


```

```
37  def command_heartbeat(obj, conn):
38      print("device {} checking in".format(obj["device_id"]))
39
40
41  def command_motion(obj, conn):
42      try:
43          device_id = obj["device_id"]
44      except KeyError:
45          print("Missing device_id")
46          return
47      print("registered motion on device {}".format(device_id))
48      if device_id in device_ids_to_pairs:
49          pair_id = device_ids_to_pairs[device_id]
50          if not "camera" in device_pairs[pair_id]:
51              print("No camera associated with this pair. no further action.")
52              return
53          camera_id = device_pairs[pair_id]["camera"]
54          print("sending capture command to device {}".format(camera_id))
55          return
56
57
58  def command_unknown(obj, conn):
59      print("Unknown command `{}`".format(obj["command"]))
60      conn.send("what?!\n".encode())
61
62
63  def quit_handler(signal, frame):
64      print("Server terminating, goodbye")
65      sys.exit(0)
66
67
68  signal.signal(signal.SIGINT, quit_handler)
69
70  listen_sock.listen(1)
71
72  while True:
73      print('Waiting for a connection...')
74      conn, client_address = listen_sock.accept()
75      try:
76          print("Connection from {}".format(client_address))
77          while True:
78              data = conn.recv(1024)
79              try:
80                  obj = json.loads(data.decode("ascii"))
81                  print(obj)
82                  command_switch = {
83                      "id": command_id,
84                      "heartbeat": command_heartbeat,
85                      "motion": command_motion
86                  }
87                  func = command_switch.get(obj["command"], command_unknown)
88                  func(obj, conn)
89              except json.JSONDecodeError:
90                  if len(data) == 0:
```

```
                    break
                print("RX [{}] {}".format(datetime.utcnow().isoformat(), data))
            except:
                print("Some random error: {}".format(sys.exc_info()[0]))
                raise
        else:
            print("No more data")
            break
    except ConnectionResetError:
        print("Connection lost.")
    finally:
        conn.close()
```

D.2 camera_base/Camera_base/camera_base.c

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <inttypes.h>
#include <linux/spi/spidev.h>
#include <letmecreate/letmecreate.h>
#include <letmecreate/core/gpio.h>

const uint8_t __CMD_WRITE_REG = 2;
const uint8_t __CMD_READ_REG = 3;
const uint8_t __CMD_GET_FRAME = 4;
const uint8_t __CMD_GET_FRAME_BUFFERED = 5;
const uint8_t __CMD_SET_ROW_NUM = 6;
const uint8_t __CMD_SET_ROW_SIZE = 7;

#define QCIF_WIDTH 176
#define QCIF_HEIGHT 144
#define FRM_WIDTH QCIF_WIDTH
#define FRM_HEIGHT QCIF_HEIGHT

uint8_t bmp_header[10] = {0x10, 0x00, FRM_HEIGHT >> 8, FRM_HEIGHT, FRM_WIDTH >> 8, FRM_WIDTH, 0xFF, 0xFF,

#ifndef MIKROBUS_IDX
#define MIKROBUS_IDX MIKROBUS_2
#endif

int8_t ready_pin;

int ready_pin_init()
{
    if (MIKROBUS_IDX == MIKROBUS_1)
    {
        ready_pin = MIKROBUS_1_INT;
    }
    else if (MIKROBUS_IDX == MIKROBUS_2)
    {
        ready_pin = MIKROBUS_2_INT;
    }
```

```
38      if (gpio_init(ready_pin) < 0)
39      {
40          return -1;
41      }
42      return 0;
43  }
44
45  int check_ready_pin()
46  {
47      uint8_t val = 2;
48      uint8_t res = gpio_get_value(ready_pin, &val);
49      while (res == 0 && (val == 0))
50      {
51          res = gpio_get_value(ready_pin, &val);
52      }
53      if (res != 0)
54      {
55          return -1;
56      }
57      if (val == 1)
58      {
59          return 0;
60      }
61      return -1;
62  }
63
64  void send_command(uint8_t command)
65  {
66      while (check_ready_pin() != 0)
67      {
68      }
69      printf("send command\n");
70      int res = spi_transfer(&command, NULL, 1);
71      printf("result: %d\n", res);
72  }
73
74  void read_bytes(uint8_t *ptr, unsigned long num)
75  {
76      while (check_ready_pin() != 0)
77      {
78      }
79      uint8_t *zeros = (uint8_t *)calloc(1, num);
80      printf("reading %d bytes\n", num);
81      spi_transfer(zeros, ptr, num);
82  }
83
84  void send_bytes(uint8_t *ptr, uint8_t num)
85  {
86      while (check_ready_pin() != 0)
87      {
88      }
89      spi_transfer(ptr, NULL, num);
90  }
91
```

```
92  uint8_t read_reg(uint8_t reg, uint8_t *val)
93  {
94      uint8_t status;
95
96      send_command(__CMD_READ_REG);
97      send_bytes(&reg, 1);
98      read_bytes(&status, 2);
99      *val = status & 0xFF;
100     return status >> 8;
101 }
102
103 uint8_t *get_camera_data()
104 {
105     uint8_t *res = (uint8_t *)calloc(1, FRM_WIDTH * FRM_HEIGHT * 2);
106     // uint8_t *res = (uint8_t *)calloc(1, 10 + (FRM_WIDTH * FRM_HEIGHT * 2));
107     if (check_ready_pin() != 0)
108     {
109         return NULL;
110     }
111     // for (int i = 0; i < 10; i++)
112     // {
113     //     res[i] = bmp_header[i];
114     // }
115     send_command(__CMD_GET_FRAME);
116     // read_bytes(res + 10, FRM_WIDTH * FRM_HEIGHT * 2);
117     read_bytes(res, FRM_WIDTH * FRM_HEIGHT * 2);
118     return res;
119 }
120
121 int main(void)
122 {
123     FILE *output = fopen("./test.bmp", "w");
124     printf("initialising spi...\n");
125     spi_init();
126     spi_select_bus(MIKROBUS_IDX);
127     spi_set_speed(MIKROBUS_IDX, SPI_10M93);
128     if (spi_set_mode(MIKROBUS_IDX, 3))
129     {
130         printf("Error setting SPI mode\n");
131         return 1;
132     }
133
134     printf("initialising ready pin...\n");
135     if (ready_pin_init() < 0)
136     {
137         fprintf(stderr, "can't open ready pin\n");
138         return 1;
139     }
140
141     printf("attempt to read reg... expecting %d\n", 0x7f);
142     uint8_t reg_val = 0;
143     read_reg(0x1c, &reg_val);
144     printf("...result: %d\n", reg_val);
145     printf("Attempting to get camera image...\n");
```

```
146     printf("god bless this mess.\n");
147     uint8_t *res = get_camera_data();
148     if (res == NULL)
149     {
150         printf("something went wrong :(\n");
151         return 2;
152     }
153     fwrite(res, 1, 10 + (FRM_WIDTH * FRM_HEIGHT * 2), output);
154 
155     free(res);
156     fclose(output);
157     spi_release();
158     return 0;
159 }
```

D.3 detect/detect.c

```
1  #include "contiki.h"
2  #include "contiki-lib.h"
3  #include "contiki-net.h"
4  
5  #include <sys/clock.h>
6  #include "letmecreate/core/network.h"
7  #include "letmecreate/core/common.h"
8  #include "letmecreate/click/motion.h"
9  //#include "letmecreate/core/debug.h"
10 #include <sys/etimer.h>
11 #include <leds.h>
12 
13 #include <string.h>
14 #include <stdio.h>
15 #include <stdbool.h>
16 
17 #define UDP_CONNECTION_ADDR "fe80:0:0:0:19:f5ff:fe89:1af0"
18 #define SERVER_PORT 9876
19 #define CLIENT_PORT 3001
20 #define BUFFER_SIZE 128
21 #define PROC_INTERVAL 20 * CLOCK_SECOND
22 
23 #ifndef DEVICE_ID
24 #define DEVICE_ID -1
25 #endif
26 
27 #ifndef PAIR_ID
28 #define PAIR_ID -1
29 #endif
30 
31 static bool motion_detected;
32 
33 static void motion_callback(uint8_t event)
34 {
35   motion_detected = true;
```

```
}

PROCESS(detect_main, "Main process for detector");
AUTOSTART_PROCESSES(&detect_main);
PROCESS_THREAD(detect_main, ev, data)
{
  PROCESS_BEGIN();
  {
    static struct etimer et;
    static struct uip_conn *connection;
    static char buf[BUFFER_SIZE];
    static int res = 0;

    etimer_set(&et, CLOCK_SECOND * 2);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

    leds_off(LED1);
    leds_on(LED2);

    connection = tcp_new_connection(SERVER_PORT, UDP_CONNECTION_ADDR);
    PROCESS_WAIT_TCP_CONNECTED();
    leds_on(LED1);
    leds_off(LED2);
    motion_detected = false;

    motion_click_enable(MIKROBUS_1);
    motion_click_attach_callback(MIKROBUS_1, motion_callback);

    etimer_set(&et, PROC_INTERVAL);
    while (true) {
      res = 0;
      PROCESS_YIELD();
      if (motion_detected) {
        sprintf(buf, "{\"device_id\":%d,\"command\":\"motion\"}", DEVICE_ID);
        tcp_packet_send(connection, buf, strlen(buf));
        PROCESS_WAIT_TCP_SENT();
        motion_detected = false;
        continue;
      }

      if (etimer_expired(&et)) {
        leds_off(LED1);
        leds_off(LED2);
        sprintf(buf, "{\"device_id\":%d,\"command\":\"heartbeat\"}", DEVICE_ID);
        res = tcp_packet_send(connection, buf, strlen(buf));
        PROCESS_WAIT_TCP_SENT();
        if (res == -1) {
          leds_on(LED2);
        }
        leds_on(LED1);
        etimer_restart(&et);
      }
    }

```

```
    motion_click_disable(MIKROBUS_1);
  }
  PROCESS_END();
}
```

D.4 classify/getimages.sh

```
#!/bin/bash

find_images() {
  while read id; do
    # download image and resize
    url=$(grep "$id" all_images.csv | head -n1 | cut -d',' -f2 | tr -d '",')
    wget "https://snapshotserengeti.s3.msi.umn.edu/$url" -O images/$id.jpg
    convert -thumbnail '128x128!' images/$id.jpg images/$id.jpg
  done
}

test -e
mkdir images
wget https://datadryad.org/bitstream/handle/10255/dryad.76010/gold_standard_data.csv
wget https://datadryad.org/bitstream/handle/10255/dryad.86392/all_images.csv
# limit to single species, one individual in frame
head -n1 gold_standard_data.csv > testdata.csv
awk -F',' 'BEGIN {OFS=","} $2 == 1 && $4 == 1 { print }' gold_standard_data.csv >> testdata.csv
# go get em
cat testdata.csv | cut -d',' -f1 | tail -n+2 | find_images
```

D.5 srv/index.js

```
const Koa = require("koa");
const objection = require("objection");
const Knex = require("knex");
const logger = require("koa-logger");
const mount = require("koa-mount");
const bodyParser = require("koa-bodyparser");

const knexConfig = require("./knexfile");

require("dotenv").config();

process.env.NODE_ENV = process.env.NODE_ENV || "development";

const knex =
  process.env.NODE_ENV === "test"
    ? require("./test/db")
    : Knex(knexConfig[process.env.NODE_ENV]); // eslint-disable-line import/newline-after-import
objection.Model.knex(knex);

const app = new Koa();

```

```
22  app.use(logger());
23  app.use(bodyParser());
24
25  // adds ability to pretty-print output if requested, using the
26  // `prettyprint` query parameter
27  app.use(async (ctx, next) => {
28    await next();
29    if (ctx.status !== 204 && ctx.query.prettyprint === "true") {
30      ctx.body = JSON.stringify(ctx.body, "\n", 2);
31    }
32  });
33
34  app.use(async (ctx, next) => {
35    await next();
36    if (ctx.status === 204) {
37      ctx.body = null;
38    } else {
39      ctx.body = { error: ctx.error, content: ctx.body };
40    }
41  });
42
43  app.use(async (ctx, next) => {
44    try {
45      await next();
46    } catch (err) {
47      ctx.body = undefined;
48      ctx.error = err.message || "An error occured";
49      ctx.status = err.status || 500;
50    }
51  });
52
53  // apps
54  app.use(mount("/basestation", require("./baseStation")));
55  app.use(mount("/sensorpair", require("./sensorPair")));
56
57  app.use(async ctx => {
58    ctx.throw("URL not found", 404);
59  });
60
61  if (process.env.NODE_ENV !== "test") {
62    app.listen(process.env.PORT || 3000, () => {
63      console.log("Server is listening.");
64    });
65  }
66
67  module.exports = app;
```

D.6 srv/baseStation.js

```
1  const Koa = require("koa");
2  const { ValidationError } = require("objection");
3  const Router = require("koa-router");
```

```
4  const { BaseStation } = require("./models");
5
6  const app = new Koa();
7
8  const router = new Router();
9
10 /**
11  * Middleware that fetches a base station using the id parameter found in the url
12  * @param {Koa.context} ctx The current app context
13  * @param {Function} next Next function in middleware chain
14  */
15 const getBaseStation = async (ctx, next) => {
16   const { id } = ctx.params;
17   const baseStation = await BaseStation.query().findById(id);
18   if (!baseStation) {
19     ctx.throw("No base station found with that id", 404);
20   }
21   ctx.state.baseStation = baseStation;
22   await next();
23 };
24
25 router.get("/:id/sensorpairs", getBaseStation, async ctx => {
26   ctx.body = await ctx.state.baseStation.$relatedQuery("sensorPairs");
27 });
28
29 router.post("/:id/sensorpairs", getBaseStation, async ctx => {
30   try {
31     const sensorPair = await ctx.state.baseStation
32       .$relatedQuery("sensorPairs")
33       .insert(ctx.request.body);
34     ctx.status = 201;
35     ctx.body = sensorPair;
36   } catch (err) {
37     ctx.body = {};
38     if (err instanceof ValidationError) {
39       ctx.throw(
40         typeof err.message === "object"
41           ? JSON.stringify(err.message)
42           : err.message,
43         400,
44       );
45     }
46     ctx.throw(err.message, 500);
47   }
48 });
49
50 router.get("/:id", getBaseStation, async ctx => {
51   ctx.body = ctx.state.baseStation.toJSON();
52 });
53
54 router.patch("/:id", async ctx => {
55   const { id } = ctx.params;
56   let baseStation;
57   try {
```

```
    baseStation = await BaseStation.query().patchAndFetchById(
      id,
      ctx.request.body,
    );
  } catch (err) {
    if (err instanceof ValidationError) {
      ctx.throw(
        typeof err.message === "object"
          ? JSON.stringify(err.message)
          : err.message,
        400,
      );
    }
    ctx.throw(err.message, 500);
  }
  if (!baseStation) {
    ctx.throw("base station not found with this id", 404);
  } else {
    ctx.body = baseStation.toJSON();
  }
});

router.delete("/:id", async ctx => {
  const { id } = ctx.params;
  const rows = await BaseStation.query().deleteById(id);
  if (rows < 1) {
    ctx.throw("base station not found with that id", 404);
  }
  ctx.status = 204;
});

router.get("/", async ctx => {
  ctx.body = await BaseStation.query();
});

router.post("/", async ctx => {
  try {
    const baseStation = await BaseStation.query().insert(ctx.request.body);
    ctx.body = baseStation;
    ctx.status = 201;
  } catch (err) {
    ctx.body = {};
    if (err instanceof ValidationError) {
      ctx.throw(
        typeof err.message === "object"
          ? JSON.stringify(err.message)
          : err.message,
        400,
      );
    }
    ctx.throw(err.message, 500);
  }
});

```

```
app.use(router.routes());
app.use(router.allowedMethods());

module.exports = app;
```

D.7 srv/sensorPair.js

```
const Koa = require("koa");
const Router = require("koa-router");
const { ValidationError } = require("objection");
const { SensorPair } = require("./models");

const app = new Koa();

const router = new Router();

/**
 * Middleware that fetches a sensor pair using the id parameter found in the url
 * @param {Koa.context} ctx The current app context
 * @param {Function} next Next function in middleware chain
 */
const getSensorPair = async (ctx, next) => {
  const { pairid } = ctx.params;
  const sensorPair = await SensorPair.query().findById(pairid);
  if (!sensorPair) {
    ctx.throw("No sensor pair found with that id", 404);
  }
  ctx.state.sensorPair = sensorPair;
  await next();
};

router.get("/:pairid/reading/", getSensorPair, async ctx => {
  ctx.body = await ctx.state.sensorPair.$relatedQuery("readings");
});

router.post("/:pairid/reading/", getSensorPair, async ctx => {
  ctx.assert(
    ctx.request.body instanceof Array,
    400,
    "You must send readings as an array",
  );
  try {
    ctx.body = await ctx.state.sensorPair
      .$relatedQuery("readings")
      .insert(ctx.request.body);
    ctx.status = 201;
  } catch (err) {
    ctx.body = {};
    if (err instanceof ValidationError) {
      ctx.throw(
        typeof err.message === "object"
          ? JSON.stringify(err.message)
```

```
          : err.message,
        400,
      );
    }
    ctx.throw(err.message, 500);
  }
});

router.get("/:pairid", getSensorPair, async ctx => {
  ctx.body = ctx.state.sensorPair.toJSON();
});

router.patch("/:pairid", async ctx => {
  const { pairid } = ctx.params;
  const sensorPair = await SensorPair.query().patchAndFetchById(
    pairid,
    ctx.request.body,
  );
  ctx.body = sensorPair;
});

router.delete("/:pairid", async ctx => {
  const { pairid } = ctx.params;
  const rows = await SensorPair.query().deleteById(pairid);
  if (rows < 1) {
    ctx.throw("No sensor pair found with that id", 404);
  }
  ctx.status = 204;
});

router.get("/", async ctx => {
  ctx.body = await SensorPair.query();
});

app.use(router.routes());
app.use(router.allowedMethods());

module.exports = app;
```

D.8 srv/knexfile.js

```
// Update with your config settings.

module.exports = {
  development: {
    client: "sqlite3",
    connection: {
      filename: "./dev.sqlite3",
    },
    useNullAsDefault: true,
  },

```

```
  staging: {
    client: "postgresql",
    connection: {
      database: "my_db",
      user: "username",
      password: "password",
    },
    pool: {
      min: 2,
      max: 10,
    },
    migrations: {
      tableName: "knex_migrations",
    },
  },

  production: {
    client: "postgresql",
    connection: {
      database: "my_db",
      user: "username",
      password: "password",
    },
    pool: {
      min: 2,
      max: 10,
    },
    migrations: {
      tableName: "knex_migrations",
    },
  },
};
```

D.9 srv/models/BaseStation.js

```
/* eslint no-param-reassign: 0 */
const { Model } = require("objection");
const uuid = require("uuid");

module.exports = class BaseStation extends Model {
  static get tableName() {
    return "basestations";
  }

  static get jsonSchema() {
    return {
      type: "object",
      properties: {
        id: { type: "string", format: "uuid" },
        name: { type: "string" },
        lat: { type: "number", default: 0 },
        lng: { type: "number", default: 0 },
```

```
      },
    };
  }

  static get relationMappings() {
    const SensorPair = require("./SensorPair");
    return {
      sensorPairs: {
        relation: Model.HasManyRelation,
        modelClass: SensorPair,
        join: {
          from: "basestations.id",
          to: "sensorpairs.basestation_id",
        },
      },
    };
  }

  async $beforeInsert() {
    this.created_at = new Date().toISOString();
    this.updated_at = this.created_at;
    this.id = uuid.v4();
  }

  $beforeUpdate() {
    this.updated_at = new Date().toISOString();
  }

  $beforeValidate(jsonSchema, json) {
    json.id = this.id;
    return jsonSchema;
  }
};
```

D.10 srv/models/Reading.js

```
const { ValidationError } = require("objection");

/* eslint class-methods-use-this: 0 */
/* eslint no-param-reassign: 0 */
const moment = require("moment");
const { Model } = require("objection");
const uuid = require("uuid");

module.exports = class Reading extends Model {
  static get tableName() {
    return "readings";
  }

  static get jsonSchema() {
    return {
      type: "object",
```

```
        required: ["t", "counts"],
        properties: {
          id: { type: "string", format: "uuid" },
          t: {
            type: "string",
            format: "date-time",
          },
          counts: {
            type: "array",
            items: {
              type: "object",
              required: ["species_id", "count"],
              properties: {
                species_id: {
                  type: "integer",
                  minimum: 0,
                },
                count: {
                  type: "integer",
                  minimum: 0,
                },
              },
            },
          },
        },
      };
    }

    static get relationMappings() {
      const SensorPair = require("./SensorPair");
      return {
        source: {
          relation: Model.BelongsToOneRelation,
          modelClass: SensorPair,
          join: {
            from: "readings.sensorpair_id",
            to: "sensorpairs.id",
          },
        },
      };
    }

    $beforeValidate(jsonSchema, json) {
      if (json.t) {
        try {
          const newDate = moment(json.t).toISOString();
          if (!newDate) {
            throw new Error("invalid date");
          }
          json.t = newDate;
        } catch (err) {
          throw Model.createValidationError({
            type: "ModelValidation",
            message: "t must be a properly-formatted date",
```

```
            data: {
              t: [
                {
                  keyword: "format",
                  message: "must be a properly-formatted time string",
                },
              ],
            },
          });
        }
      }

      return super.$beforeValidate(jsonSchema, json);
    }

    $afterValidate(json, opt) {
      json.counts = json.counts.filter(el => el.count > 0);

      // check for duplicate species_ids
      const ids = json.counts.map(el => el.species_id);
      if (new Set(ids).size !== ids.length) {
        throw Model.createValidationError({
          type: "modelValidation",
          message: "each `counts` item must have a unique species_id",
          data: {
            counts: [
              {
                keyword: "format",
                message: "each item must have a unique species_id",
              },
            ],
          },
        });
      }
      return super.$afterValidate(json, opt);
    }

    async $beforeInsert() {
      this.created_at = new Date().toISOString();
      this.updated_at = this.created_at;
      this.id = uuid.v4();
    }

    $beforeUpdate() {
      this.updated_at = new Date().toISOString();
    }
  };
```

D.11 srv/models/SensorPair.js

```
  /* eslint-disable class-methods-use-this */
  const { ValidationError } = require("objection");
```

```
3
4  const { Model } = require("objection");
5  const uuid = require("uuid");
6
7  module.exports = class SensorPair extends Model {
8    static get tableName() {
9      return "sensorpairs";
10   }
11
12   static get jsonSchema() {
13     return {
14       type: "object",
15       required: ["camera_id", "motion_id"],
16       properties: {
17         id: { type: "string", format: "uuid", readOnly: true },
18         lat: { type: "number", default: 0 },
19         lng: { type: "number", default: 0 },
20         name: { type: "string" },
21         camera_id: { type: "integer", minimum: 0 },
22         motion_id: { type: "integer", minimum: 0 },
23       },
24     };
25   }
26
27   static get relationMappings() {
28     const BaseStation = require("./BaseStation");
29     const Reading = require("./Reading");
30     return {
31       baseStation: {
32         relation: Model.BelongsToOneRelation,
33         modelClass: BaseStation,
34         join: {
35           from: "sensorpairs.basestation_id",
36           to: "basestations.id",
37         },
38       },
39       readings: {
40         relation: Model.HasManyRelation,
41         modelClass: Reading,
42         join: {
43           from: "sensorpairs.id",
44           to: "readings.sensorpair_id",
45         },
46       },
47     };
48   }
49
50   async $beforeInsert() {
51     this.created_at = new Date().toISOString();
52     this.updated_at = this.created_at;
53     this.id = uuid.v4();
54   }
55
56   $beforeUpdate() {
```

```
      this.updated_at = new Date().toISOString();
    }

    $beforeValidate(jsonSchema, json) {
      // both coords set, or none. no inbetween!
      if ((json.lat && !json.lng) || (json.lng && !json.lat)) {
        throw new ValidationError({ statusCode: 400, type: "ModelValidation" });
      }
      return jsonSchema;
    }
  };
```