

LONDON'S GLOBAL UNIVERSITY



Tracking Wildlife Counts Using the Internet Of Things

Matthew Bell¹

`m.bell@cs.ucl.ac.uk`

BSc Computer Science

Supervisor: Dr Kevin Bryson

Submission date: Day Month Year

¹**Disclaimer:** This report is submitted as part requirement for the Computer Science BSc 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. 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 hours of time when conducting studies using camera traps.

The project also explores various methods in which species of animals could be identified automatically, given various constraints of how the system can work, including the availability and speed of the network link.

To-Do (Delete Before Submission)

	include graphic of camera pairs in trees etc?	7
	Include image of 6LoWPAN clicker	8
	Add list of commands?	8
	include image of motion click	8
	any other events?	8
	include image of camera click	9
	include E-R diagram (and ref here)	10

Contents

1	Introduction and background	4
1.1	Motivation and need	4
1.2	Requirements	4
1.3	Research and Literature Review	5
2	Design	6
2.1	System Architecture	6
2.1.1	Base Station	6
2.1.2	Remote Sensors	7
2.1.3	Web Server	9
3	Implementation	11
4	Testing	12
4.1	API Server Testing	12
4.1.1	Testing Framework	12
4.1.2	Creating The Unit Tests	13
4.1.3	Creating The Integration Tests	13
4.1.4	Test Results And Coverage Report	13
5	Conclusion	15
A	Glossaries	16
	Glossary	16
	Acronyms	16
B	Bibliography	18
C	API Specification	19
C.1	Base Station	19
C.1.1	Get a list of all base stations	19
C.1.2	Add a new base station	20
C.1.3	Modify an existing base station	20
C.1.4	Delete an existing base station	21
C.2	Base Station Sensor Pairs	21

C.2.1	Get a list of a base station's sensor pairs	21
C.2.2	Add a new sensor pair to a base station	22
C.3	Sensor Pair	22
C.3.1	Get a list of all sensor pairs	22
C.3.2	Modify an existing sensor pair	23
C.3.3	Delete an existing sensor pair	24
C.4	Sensor Pair Reading	24
C.4.1	Get a list of all readings for a sensor	24
C.4.2	Add one or more readings to a sensor pair	25
D	Code Listing	26
D.1	base/___main___ .py	26
D.2	detect/detect.c	28
D.3	classify/classify_image.c	31

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 [11]. 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 [15], 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” [13]. 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 [4]. 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.

1.3 Research and Literature Review

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.

Chapter 2

Design

2.1 System Architecture

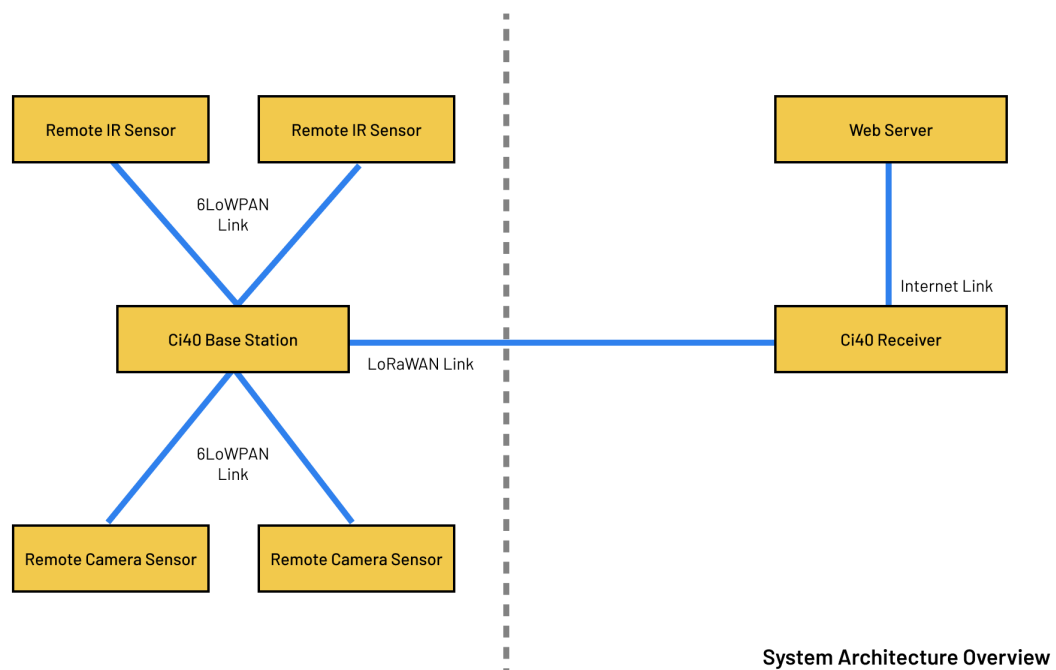


Figure 2.1: System architecture overview, showing how the high-level components of the system are connected.

Figure 2.1 shows the connections between the high-level components in the system, including the sensor devices 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 [13]. It is a development board that is highly specialised for rapidly building Internet of

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 [3]. 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 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

include graphic of camera pairs in trees etc?

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 [4], with either a camera or motion detector attachment.

Hardware Design

Include image of 6LoWPAN clicker

The *MikroElektronika* 6LoWPAN Click Board, as previously mentioned, runs a Real-Time Operating System (RTOS) called Contiki. According to the Contiki project website [2], it is ideal for low-power IoT projects since it supports the full IPv4 and IPv6 networking suites, as well as being able to run on “tiny systems, only having a few kilobytes of memory available”. *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 [14].

Creator also provide a set of guides to help set up and write programs to the flash memory onboard the clicker device [12], 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.

Add list of commands?

Motion Detector Sensor

include image of motion click

The motion detector sensor uses the 6LoWPAN Click board described above, with a *MikroElektronika* Motion Click device [7] integrated onto the board using the “mikroBUS” port. According to the product page [7], 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 [14].

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.

any other events?

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 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

include image of camera click

The camera sensor comprises of the same 6LoWPAN Clicker board, but instead of a motion sensor, there is a *MikroElektronika* Camera Click board [5] 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* [6] 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.

include E-R diagram (and ref here)

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

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 [10] 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 [9].

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

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* [16], but combined these metrics serve as a useful insight into how effectively the tests actually test the code written.

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.

Chapter 5

Conclusion

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. 5, 7–9

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. 4, 7

M

mikroBUS Bus standard for integrating IoT sensors onto development boards. Contains pins for SPI, analog data transmission, power, and an interrupt pin. 8, 9

MIPS Multiprocessor without Interlocked Pipeline Stages, a type of processor architecture. 5

O

opcode Short for operation code, a command or instruction that may be part of a device’s instruction list. 9

Acronyms

A

API Application Programming Interface. 2, 4, 9, 10, 12, 13

B

BDD Behaviour-Driven Development. 12

H

HTTP Hyper-Text Transfer Protocol. 10, 12

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, 10

J

JSON JavaScript Object Notation. 8

R

RTOS Real-Time Operating System. 5, 8

S

SPI Serial Peripheral Interface bus. 9

T

TCP Transmission Control Protocol. 8, 9

TDD Test-Driven Development. 13

Appendix B

Bibliography

- [1] Airbnb. *Airbnb JavaScript Style Guide()*. URL: <https://github.com/airbnb/javascript/> (visited on 04/23/2018).
- [2] Contiki. *Contiki: The Open Source Operating System for the Internet of Things*. URL: <http://www.contiki-os.org> (visited on 04/13/2018).
- [3] David Culler and Samita Chakrabarti. “6LoWPAN: Incorporating IEEE 802.15. 4 into the IP architecture”. In: *IPSO Alliance, White paper* (2009).
- [4] MikroElektronika d.o.o. *6LoWPAN clicker - a compact development board / MikroElektronika*. URL: <https://www.mikroe.com/clicker-6lowpan> (visited on 04/12/2018).
- [5] 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).
- [6] MikroElektronika d.o.o. *LibStock - Camera Click - Example*. URL: <https://libstock.mikroe.com/projects/view/1263/camera-click-example> (visited on 04/15/2018).
- [7] 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).
- [8] 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).
- [9] Facebook Inc. *Expect · Jest*. URL: <https://facebook.github.io/jest/docs/en/expect.html> (visited on 04/23/2018).
- [10] Facebook Inc. *Jest • Delightful JavaScript Testing*. URL: <https://facebook.github.io/jest/> (visited on 04/23/2018).
- [11] 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.
- [12] Imagination Technologies Limited. URL: <https://docs.creatordev.io/clicker/guides/quick-start-guide/> (visited on 04/13/2018).
- [13] 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).

- [14] Imagination Technologies Limited. *CreatorDev/LetMeCreate*. URL: <https://github.com/CreatorDev/LetMeCreate> (visited on 04/13/2018).
- [15] 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.
- [16] 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": "000000000-0000-0000-0000-000000000000",
      "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": "000000000-0000-0000-0000-000000000000",
    "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": "000000000-0000-0000-0000-000000000000",

```

```

    "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": "000000000-0000-0000-0000-000000000000",
      "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": "000000000-0000-0000-0000-000000000000",
    "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": "000000000-0000-0000-0000-000000000000",
      "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": "000000000-0000-0000-0000-000000000000",
    "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": "000000000-0000-0000-0000-000000000000",
      "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": "000000000-0000-0000-0000-000000000000",
      "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

```
1  #
2  # A very simple UDP server.
3  # Just listens to port 9876 and prints to STDOUT
4  # Plays nicely with `tee` as a result.
5  import signal
6  import socket
7  import sys
8  from datetime import datetime
9  import json
10
11 listen_sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)
12 listen_sock.bind(("::", 9876))
13
14 device_pairs = {}
15
16 # Maps device ids to their pair ids
17 device_ids_to_pairs = {}
18
19
20 def command_id(obj, conn):
21     try:
22         pair_id = obj["pair_id"]
23         device_id = obj["device_id"]
24         device_type = obj["type"]
25     except KeyError:
26         print("missing keys. make sure `pair_id`, `device_id` and `type` are provided")
27         return
28     print("device {} to be paired in pair {}".format(device_id, pair_id))
```

```

29     if not pair_id in device_pairs:
30         device_pairs[pair_id] = {}
31     if not device_id in device_ids_to_pairs:
32         device_ids_to_pairs[device_id] = device_id
33     device_pairs[pair_id][device_type] = device_id
34     print("device pair {}: now {}".format(pair_id, device_pairs[pair_id]))
35
36
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:
91                     break
92                 print("RX [{}] {}".format(datetime.utcnow().isoformat(), data))
93             except:
94                 print("Some random error: {}".format(sys.exc_info()[0]))
95                 raise
96         else:
97             print("No more data")
98             break
99     except ConnectionResetError:
100         print("Connection lost.")
101     finally:
102         conn.close()

```

D.2 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 "letmcreate/core/network.h"
7 #include "letmcreate/core/common.h"
8 #include "letmcreate/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;
36 }
37
38 PROCESS(detect_main, "Main process for detector");
39 AUTOSTART_PROCESSES(&detect_main);
40 PROCESS_THREAD(detect_main, ev, data)
41 {
42     PROCESS_BEGIN();
43     {
44         static struct etimer et;
45         static struct uip_conn *connection;
46         static char buf[BUFFER_SIZE];
47         static int res = 0;
48
49         etimer_set(&et, CLOCK_SECOND * 2);
50         PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
51

```



```

52     leds_off(LED1);
53     leds_on(LED2);
54
55     connection = tcp_new_connection(SERVER_PORT, UDP_CONNECTION_ADDR);
56     PROCESS_WAIT_TCP_CONNECTED();
57     leds_on(LED1);
58     leds_off(LED2);
59     motion_detected = false;
60
61     motion_click_enable(MIKROBUS_1);
62     motion_click_attach_callback(MIKROBUS_1, motion_callback);
63
64     etimer_set(&et, PROC_INTERVAL);
65     while (true) {
66         res = 0;
67         PROCESS_YIELD();
68         if (motion_detected) {
69             sprintf(buf, "{\"device_id\":%d,\"command\": \"motion\"}", DEVICE_ID);
70             tcp_packet_send(connection, buf, strlen(buf));
71             PROCESS_WAIT_TCP_SENT();
72             motion_detected = false;
73             continue;
74         }
75
76         if (etimer_expired(&et)) {
77             leds_off(LED1);
78             leds_off(LED2);
79             sprintf(buf, "{\"device_id\":%d,\"command\": \"heartbeat\"}", DEVICE_ID);
80             res = tcp_packet_send(connection, buf, strlen(buf));
81             PROCESS_WAIT_TCP_SENT();
82             if (res == -1) {
83                 leds_on(LED2);
84             }
85             leds_on(LED1);
86             etimer_restart(&et);
87         }
88     }
89
90     motion_click_disable(MIKROBUS_1);
91 }
92 PROCESS_END();
93 }

```

D.3 classify/classify_image.c

```
1  # Copyright 2015 The TensorFlow Authors. All Rights Reserved.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #     http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14 # =====
15
16 """Simple image classification with Inception.
17
18 Run image classification with Inception trained on ImageNet 2012 Challenge data
19 set.
20
21 This program creates a graph from a saved GraphDef protocol buffer,
22 and runs inference on an input JPEG image. It outputs human readable
23 strings of the top 5 predictions along with their probabilities.
24
25 Change the --image_file argument to any jpg image to compute a
26 classification of that image.
27
28 Please see the tutorial and website for a detailed description of how
29 to use this script to perform image recognition.
30
31 https://tensorflow.org/tutorials/image\_recognition/
32 """
33
34 from __future__ import absolute_import
35 from __future__ import division
36 from __future__ import print_function
37
38 import argparse
39 import os.path
40 import re
41 import sys
```

```

42 import tarfile
43
44 import numpy as np
45 from six.moves import urllib
46 import tensorflow as tf
47
48 FLAGS = None
49
50 # pylint: disable=line-too-long
51 DATA_URL = 'http://download.tensorflow.org/models/image/imagenet/inception-2015-12
52 # pylint: enable=line-too-long
53
54
55 class NodeLookup(object):
56     """Converts integer node ID's to human readable labels."""
57
58     def __init__(self,
59                 label_lookup_path=None,
60                 uid_lookup_path=None):
61         if not label_lookup_path:
62             label_lookup_path = os.path.join(
63                 FLAGS.model_dir, 'imagenet_2012_challenge_label_map_proto.pbtxt')
64         if not uid_lookup_path:
65             uid_lookup_path = os.path.join(
66                 FLAGS.model_dir, 'imagenet_synset_to_human_label_map.txt')
67         self.node_lookup = self.load(label_lookup_path, uid_lookup_path)
68
69     def load(self, label_lookup_path, uid_lookup_path):
70         """Loads a human readable English name for each softmax node.
71
72         Args:
73             label_lookup_path: string UID to integer node ID.
74             uid_lookup_path: string UID to human-readable string.
75
76         Returns:
77             dict from integer node ID to human-readable string.
78         """
79         if not tf.gfile.Exists(uid_lookup_path):
80             tf.logging.fatal('File does not exist %s', uid_lookup_path)
81         if not tf.gfile.Exists(label_lookup_path):
82             tf.logging.fatal('File does not exist %s', label_lookup_path)
83
84         # Loads mapping from string UID to human-readable string

```

```

85     proto_as_ascii_lines = tf.gfile.GFile(uid_lookup_path).readlines()
86     uid_to_human = {}
87     p = re.compile(r'[n\d]*[ \S,]*')
88     for line in proto_as_ascii_lines:
89         parsed_items = p.findall(line)
90         uid = parsed_items[0]
91         human_string = parsed_items[2]
92         uid_to_human[uid] = human_string
93
94     # Loads mapping from string UID to integer node ID.
95     node_id_to_uid = {}
96     proto_as_ascii = tf.gfile.GFile(label_lookup_path).readlines()
97     for line in proto_as_ascii:
98         if line.startswith('  target_class:'):
99             target_class = int(line.split(':')[1])
100         if line.startswith('  target_class_string:'):
101             target_class_string = line.split(':')[1]
102             node_id_to_uid[target_class] = target_class_string[1:-2]
103
104     # Loads the final mapping of integer node ID to human-readable string
105     node_id_to_name = {}
106     for key, val in node_id_to_uid.items():
107         if val not in uid_to_human:
108             tf.logging.fatal('Failed to locate: %s', val)
109             name = uid_to_human[val]
110             node_id_to_name[key] = name
111
112     return node_id_to_name
113
114     def id_to_string(self, node_id):
115         if node_id not in self.node_lookup:
116             return ''
117         return self.node_lookup[node_id]
118
119
120     def create_graph():
121         """Creates a graph from saved GraphDef file and returns a saver."""
122         # Creates graph from saved graph_def.pb.
123         with tf.gfile.FastGFile(os.path.join(
124             FLAGS.model_dir, 'classify_image_graph_def.pb'), 'rb') as f:
125             graph_def = tf.GraphDef()
126             graph_def.ParseFromString(f.read())
127             _ = tf.import_graph_def(graph_def, name='')

```

```

128
129
130 def run_inference_on_image(image):
131     """Runs inference on an image.
132
133     Args:
134         image: Image file name.
135
136     Returns:
137         Nothing
138     """
139     if not tf.gfile.Exists(image):
140         tf.logging.fatal('File does not exist %s', image)
141     image_data = tf.gfile.GFile(image, 'rb').read()
142
143     # Creates graph from saved GraphDef.
144     create_graph()
145
146     with tf.Session() as sess:
147         # Some useful tensors:
148         # 'softmax:0': A tensor containing the normalized prediction across
149         # 1000 labels.
150         # 'pool_3:0': A tensor containing the next-to-last layer containing 2048
151         # float description of the image.
152         # 'DecodeJpeg/contents:0': A tensor containing a string providing JPEG
153         # encoding of the image.
154         # Runs the softmax tensor by feeding the image_data as input to the graph.
155         softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')
156         predictions = sess.run(softmax_tensor,
157                                {'DecodeJpeg/contents:0': image_data})
158         predictions = np.squeeze(predictions)
159
160         # Creates node ID --> English string lookup.
161         node_lookup = NodeLookup()
162
163         top_k = predictions.argsort()[-FLAGS.num_top_predictions:][::-1]
164         for node_id in top_k:
165             human_string = node_lookup.id_to_string(node_id)
166             score = predictions[node_id]
167             print('%s (score = %.5f)' % (human_string, score))
168
169
170 def maybe_download_and_extract():

```

```

171     """Download and extract model tar file."""
172     dest_directory = FLAGS.model_dir
173     if not os.path.exists(dest_directory):
174         os.makedirs(dest_directory)
175     filename = DATA_URL.split('/')[-1]
176     filepath = os.path.join(dest_directory, filename)
177     if not os.path.exists(filepath):
178         def _progress(count, block_size, total_size):
179             sys.stdout.write('\r>> Downloading %s %.1f%%' % (
180                 filename, float(count * block_size) / float(total_size) * 100.0))
181             sys.stdout.flush()
182         filepath, _ = urllib.request.urlretrieve(DATA_URL, filepath, _progress)
183         print()
184         statinfo = os.stat(filepath)
185         print('Successfully downloaded', filename, statinfo.st_size, 'bytes.')
186         tarfile.open(filepath, 'r:gz').extractall(dest_directory)
187
188
189     def main(_):
190         maybe_download_and_extract()
191         image = (FLAGS.image_file if FLAGS.image_file else
192                 os.path.join(FLAGS.model_dir, 'cropped_panda.jpg'))
193         run_inference_on_image(image)
194
195
196     if __name__ == '__main__':
197         parser = argparse.ArgumentParser()
198         # classify_image_graph_def.pb:
199         #   Binary representation of the GraphDef protocol buffer.
200         # imagenet_synset_to_human_label_map.txt:
201         #   Map from synset ID to a human readable string.
202         # imagenet_2012_challenge_label_map_proto.pbtxt:
203         #   Text representation of a protocol buffer mapping a label to synset ID.
204         parser.add_argument(
205             '--model_dir',
206             type=str,
207             default='/tmp/imagenet',
208             help="""\
209             Path to classify_image_graph_def.pb,
210             imagenet_synset_to_human_label_map.txt, and
211             imagenet_2012_challenge_label_map_proto.pbtxt.\
212             """
213         )

```

```

214     parser.add_argument(
215         '--image_file',
216         type=str,
217         default='',
218         help='Absolute path to image file.'
219     )
220     parser.add_argument(
221         '--num_top_predictions',
222         type=int,
223         default=5,
224         help='Display this many predictions.'
225     )
226     FLAGS, unparsed = parser.parse_known_args()
227     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```