Telemetry

Emily Horsman <horsmane@mcmaster.ca>, Tanner Ryan <ryant3@mcmaster.ca>

April 30, 2020

COMPSCI 4ZP6
Capstone Project
McMaster University

Preamble

Telemetry saves significant development time for common embedded applications.

There is a large market of hardware peripherals which use an I^{2}C bus to communicate with a computer. They usually interface with an embedded device.[1] These embedded controllers need to be flashed to change application logic and this often requires physical access. There are thousands of such peripherals including sensors that measure temperature, moisture, colour, current, etc. Hobbyists and industrial prototypers buy these sensors from distributors such as Adafruit and Sparkfun. These distributors often share libraries that allow these sensors to be used with common embedded controllers with little development time.[2] However, these libraries have a lot of redundancy amongst them and often contain technical tradeoffs.[3]

Writing a program for embedded controllers that reads and writes to I^{2}C-based hardware peripherals traditionally involves three tasks:

1. Getting data off the I^{2}C bus with sensor-specific protocols running in an "application layer".
2. Turning raw bytes from the sensor protocol into semantically sensible values. i.e., from bytes to a temperature reading in degrees Celsius.
3. Managing multiple sensors connected to the same I^{2}C bus which are read and written to at various intervals.

Telemetry is a library for embedded devices that builds on an abstraction of hardware peripherals to simplify these tasks.

1. Generalizes interactions on the I^{2}C bus to allow declarative configurations for new sensors to be written instead of application logic.

[1] "Arduino-compatible" microcontrollers are popular choices amongst hobbyists and industrial prototypers.
[2] e.g., For the SHT31 temperature and humidity sensor: `https://github.com/adafruit/Adafruit_SHT31`
[3] e.g., calling functions which sleep the device instead of using an event loop that would allow other tasks to be performed.

2. Decouples the application logic which deals with data semantics from the embedded device and its development environment. Developers will write a small program independent of the firmware flashed to embedded devices without needing to deal with low-level details. These programs can be written in any language and run on servers, instead of being written in the low-level systems languages that embedded devices typically use.[4]
3. Manages peripherals on an I^2C bus without writing additional firmware logic. This includes logic for automatically discovering and processing data when a sensor is physically connected.

This achieves meaningful outcomes for development teams.

1. Much less development time to work with data from sensors. Reduces prototyping to a "plug-and-play" experience.
2. Reduces skill specialization required for embedded development. Application logic can be written in any language without knowledge of embedded C++.
3. Telemetry can be used as a pre-packaged firmware that can be uploaded to a compatible embedded device without changes. This allows you to physically connect sensors and immediately store data and consume data from a provided API.

Emily Horsman and Tanner Ryan are responsible for this aspect of the project.

Acknowledgement

We would like to thank our supervisors Dr. Jacques Carette <carette@mcmaster.ca> and Dr. Spencer Smith <smiths@mcmaster.ca> for their guidance throughout the project. We would also like to acknowledge Dr. Frantisek Franek <franek@mcmaster.ca> for the adminstration of the capstone course.

BSD 2-Clause License

[4] C and C++ are common choices for embedded controllers. Developers can use anything — even something like Haskell — for Telemetry .

Contents

Revision History

Date	Version	Description	Author(s)
November 2, 2019	1.0	Initial document.	Emily Horsman, Tanner Ryan
November 6, 2019	1.1	Extend project background, system overview.	Emily Horsman
December 31, 2019	2.0	Add software design specification, architecture.	Emily Horsman, Tanner Ryan
April 29, 2020	2.1	Update network protocols to reflect changes in implementation. Minor corrections to software design specification. Add requested viewpoints: interface, structure, interaction, resource. Add user (application) interface specifications. Add programming guide.	Emily Horsman, Tanner Ryan

Software Requirements Specification

1 Introduction

This document is to describe the system architecture of Telemetry . Defined immediately below are terms referenced within this document.

Emily Horsman and Tanner Ryan are responsible for this aspect of the project.

2 Definitions

1. I^2C : Inter-Integrated Circuit, a standardized, packet switched serial bus. Allows for bidirectional communication between hardware peripheral and microcontrollers.
2. Microcontroller: Arduino powered, single board computer. Provides an I2Cbus and IP network connection.
3. Controller: main application server for Telemetry. Responsible for communication with microcontroller(s), exposing HTTP API service, and serving the dashboard.
4. Peripheral processor: responsible for informing the controller how a sensor model works via Telemetry's external interface.
5. Peripheral: Any I^2C hardware sensor (e.g. temperature, accelerometer, thermal cam- era) which uses the I^2C standard "register paradigm".
6. HTTP API: application programming interface, a publicly exposed HTTP endpoint to provide a defined request-response messaging system.
7. Peripheral payload: Raw data originating from a peripheral.
8. Peripheral data: List of key-values derived from peripheral payload.
9. Frame payload: Data sent between a microcontroller and a controller.

3 Context

I^2C is an existing bus protocol which enables hardware peripherals to communicate with each other and devices such as microcontrollers. Currently, repetitive application logic must be written for each I^2C device. This application logic is hard to write in a maintainable, self-documenting way. Any modification of the application logic means that firmware updates need to be applied to other devices on the I^2C bus (most commonly, the microcontroller serving as the "master" device). This creates a significant amount of work throughout the prototyping and maintenance stages of a hardware project.

4 System Overview

Telemetry is a proposed software solution to abstract the most common usages of the I^2C protocol. This layer of abstraction allows the end user to rapidly provision microcontrollers for the ingestion of peripheral data. Telemetry will provide the user with a web dashboard and a HTTP API for the configuration of peripherals and the consumption of peripheral data.

5 Project Goals

This version of Telemetry will support the following features.

5.0.1 Websocket stream

Telemetry will expose a standard Websocket stream to allow for real-time integration of peripheral data into an existing application.

5.0.2 HTTP API

Telemetry will expose an HTTP API that allows for the querying of current and past data. This provides users with a non real-time data source for integration with legacy applications.

5.0.3 Microcontroller and Peripheral Autodiscovery

Telemetry provides the user with an Arduino library and a containerized controller. The library is be responsible for:

1. Initializing and re-establishing a connection to the controller.
2. Autodiscovery of I^2C peripherals.
3. Collection and forwarding of peripheral payloads to controller.

The controller is responsible for providing the Web Dashboard and HTTP API, which the user will interact with.

6 Non-project Goals

This version of Telemetry will not support the following features.

6.1 Multitenancy

The controller of Telemetry is designed around the requirements of an individual or single organization. No user authentication is performed.

6.2 End-to-End Encryption

All frame payloads sent between microcontrollers and the controller are over plaintext. Frame payloads must not contain confidential information.

7 Design Considerations

7.1 Assumptions and Dependencies

7.1.1 Software and Hardware

This version of Telemetry 's microcontroller library will only support Arduino-compatible platforms. The Arduino ESP32 microcontroller will have tested hardware support.

7.1.2 Operating Systems

The controller will be packaged using Docker, a containerization platform. Therefore, the controller will be compatible with all operating systems that support Docker. Linux and macOS will have tested software support.

7.2 General Constraints

7.2.1 Network Connectivity

Telemetry will have the ability to work well on low bandwidth, high latency networks. Although the microcontrollers are capable of re-establishing a connection with the controller, the network should have minimal packet loss for real-time data consumption.

Microcontrollers are not required to be on the same network as the controller, although the controller must be reachable from the microcontroller's network.

8 System Design

All **bold** keywords refer to definitions or architecture components.

8.1 Architecture

The following depicts a Telemetry setup with two **microcontrollers** and three **peripheral processors**.

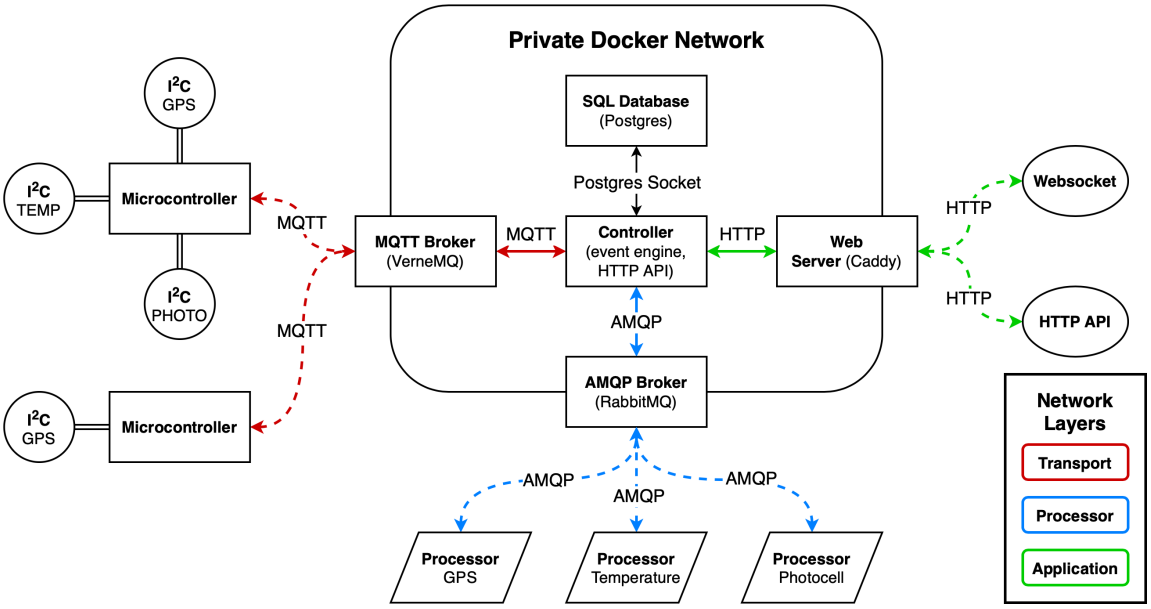

Figure 1: System Architecture

8.1.1 I²C Peripheral

A hardware sensor that utilizes the I²C "register paradigm". Connected to the **microcontroller** through direct hardwired connection.

8.1.2 Microcontroller

An Arduino powered computer. Responsible for establishing hardware connection with **peripheral(s)**. The microcontroller will run Telemetry's Arduino library.

8.1.3 MQTT Broker

Responsible for exposing an MQTT server, a lightweight publish/subscribe message transport protocol. The **microcontroller(s)** and the **processor** are responsible for establishing an MQTT connection. The **microcontroller(s)** will re-establish the MQTT connection in the event that the connection is interrupted.

8.1.4 Controller

Main event engine for Telemetry. The interfacing is broken down into four categories:

MQTT Interfacing:

1. Receive registration messages (**frame payload**) from **microcontroller(s)** over MQTT.
2. Receive **peripheral payloads** from **microcontroller(s)** over MQTT.
3. Publish provisioning message (**frame payload**) to **microcontroller(s)** over MQTT.
4. Publish one-off messages (**frame payload**) to **microcontroller(s)** over MQTT (i.e. reboot, manually poll data).

AMQP Interfacing:

1. Publish raw **peripheral payloads** over AMQP.
2. Receive **peripheral data** over AMQP.

SQL Database Interfacing:

1. Insert **peripheral data**.
2. Insert/update/delete **microcontroller** configuration.
3. Insert/update/delete **dashboard** configuration.
4. Query **peripheral data**.

Web Server Interfacing:

1. Expose HTTP endpoints for REST **API**.
2. Expose websocket connection for real-time streaming updates.

8.1.5 SQL Database

Responsible for persistent storage of **peripheral data**, **dashboard** configuration, and **microcontroller** provisioning.

8.1.6 AMQP Broker

Akin to MQTT Broker, responsible for exposing AMQP server, a publish/subscribe message transport protocol with queueing capabilities. The **processor(s)** and the **controller** are responsible for establishing an AMQP connection. The **processor(s)** will re-establish the AMQP connection in the event that the connection is interrupted.

8.1.7 Peripheral Processor

An individual **peripheral processor** is required per **peripheral** type (i.e. Adafruit temperature sensor). It must establish an AMQP connection with Telemetry 's AMQP broker. The **processor** will advertise the type of **peripheral** that it can process. It will then receive raw **peripheral payloads** over AMQP that match the peripheral type. The processor will parse the raw data into value(s), publishing the **peripheral data** over AMQP.

8.1.8 Web Server

Responsible for performing SSL (TLS) termination for the **dashboard** and HTTP **API** services.

8.1.9 Websocket

Websocket stream that user may interact with, satisfying project goals.

8.1.10 HTTP API

HTTP **API** that user or scripts may interact with, satisfying project goals.

8.2 Transport Protocol

All data transmitted over the **transport layer** will be using a custom wire (binary) format over MQTT. The following transactions are supported by Telemetry 's protocol.

8.2.1 Registration Frame

A **microcontroller** must send a registration frame when:

1. MQTT connection is established.
2. MQTT connection is re-established after disconnection.
3. A **peripheral** was connected or disconnected.

The registration frame contains:

1. Telemetry Arduino library version identifier.
2. **Microcontroller** unique identifier (MAC address).
3. **Microcontroller** network IP address.
4. List of **peripherals** (active I^2C addresses).

The registration frame is used for provisioning of **microcontrollers**. This provisioning builds a registry of all **microcontrollers** and **peripherals** connected to Telemetry .

8.2.2 Payload Frame

A **microcontroller** must send a payload frame when:

1. Raw data is collected from **peripheral**.

The payload frame contains:

1. **I²C** address of data collection.
2. Raw **peripheral payload**.

The payload frame is used for sending raw data from **peripheral(s)** to the **controller**.

8.2.3 Provisioning Frame

A **controller** must send a provisioning frame when:

1. A **registration frame** is received.
2. The provisioning of a **microcontroller** has been modified.

The provisioning frame contains:

1. **Microcontroller** schedule.

The provisioning frame is used for dynamically assigning the **microcontroller** a schedule. This schedule identifies **I²C** addresses to poll, the polling format, and the polling interval. The **microcontroller** will continuously run this schedule, publishing **payload frames** as data is collected. If a new **provisioning frame** is received by a **microcontroller**, the new schedule is to be utilized.

8.2.4 Request Frame

A **controller** must send a request frame when:

1. The user requests the **microcontroller** to reboot.
2. The user requests the **microcontroller** to manually perform one iteration of the **schedule**.

The request frame contains:

1. Action for **microcontroller** to perform.

The request frame is used for notifying the **microcontroller** to perform one-off events. The requests are activated by the user. This is for debugging purposes.

8.3 Processor Protocol

All data transmitted over the **processor layer** will be JSON over AMQP.

8.3.1 AMQP Initialization

A **peripheral processor** will interface with Telemetry by initializing an AMQP connection with the **AMQP broker**. When the connection is established, the **peripheral processor** must subscribe to the `controller.addr.#` route, where `addr` is replaced with the **peripheral** bus address which this processor is capable of parsing. The **peripheral processor** must also subscribe to `global.req` for listening to configuration requests.

Peripheral payloads matching the **peripheral** bus addressed will be pushed to this processor. If multiple **peripheral processors** of the same bus address are connected, **peripheral payloads** will be distributed in a round-robin fashion.

8.3.2 Consume Peripheral Payload

When new data is collected from a **peripheral** with bus address `x`, connected to a microcontroller with MAC address `y`, the raw **peripheral payload** will be received on a **peripheral processor** subscribed to `controller.x.y`. The **peripheral processor** must convert the raw, often proprietary, **I²C** data into a list of key and values, that can be ingested by Telemetry .

The incoming data will be in the following JSON format:

```
{
  "busId": 1,
  "busAddr": 68,
  "data": "MjAyMC0wNC0yOVQwMjoxMToyNFo="
}
```

The `busId` and `busAddr` will be modified accordingly. The `data` field is the base64 representation of the raw data collected from the peripheral. This raw data must be converted into a set of key-values and published as described below.

The **peripheral processor** may be written in any language. The only requirement is that the language must have support for AMQP and JSON.

8.3.3 Publish Peripheral Data

When the **peripheral processor** has generated a JSON object of keys and values, the JSON data will be published over AMQP with a routing key of `data.x.y`, where both `x` and `y` are equal to the variables during **peripheral payload** consumption.

Telemetry will insert this data into the database, making it available on the **dashboard** and HTTP **API** service.

8.3.4 Configuration Advertisement

On **peripheral processor** initialization, and when an empty message is received on the `global.req` route, the configuration profile must be published on the `global.config` route. The configuration profile must be in the following JSON format:

```
{
  "busAddr": 68,
  "name": "SHT31",
  "readDefinitions": [
    {
      "definitionId": 1,
      "registerIdLength": 16,
      "registerId": 9216,
      "registerBlockLength": 1,
      "numBytesPerRegister": 6,
      "readPeriod": 500
    }
  ]
}
```

Replace above variables as described in a **peripheral's** specification document. If a **peripheral** does not have this information available, it is considered incompatible with Telemetry .

8.4 Application Protocol

The standard REST API pattern is to be utilized for the HTTP **API** service. All data requested and returned by **API** is to be in valid JSON format.

9 Languages and Frameworks

9.1 Frontend: React

React is an open source JavaScript framework, allowing for the creation of interactive, modular user interfaces. React allows us to write a lightweight user interface in a declarative fashion with modern functional-style JavaScript. Typed JavaScript has good support for React. The **dashboard** includes many state transitions (e.g., **microcontrollers** and **peripherals** coming online and offline) which are better handled in a declarative fashion than an imperative one, where synchronization bugs are common. The React community offers many well-written open-source "batteries" for data visualization that will aide in a rapid development process.

9.2 Transport Layer: MQTT

MQTT is a low-overhead messaging protocol. MQTT enables Telemetry to reliably and efficiently collected data from remote Arduino **microcontrollers**. It ensures that data sent between **microcontroller(s)** and the **controller** is delivered. Such delivery reports are essential on networks that contain packet loss.

VerneMQ is an open source, industry standard MQTT broker.

9.3 Processor Layer: AMQP

AMQP is very similar to MQTT, except for the addition of queuing capabilities. Message queuing is used for Telemetry to process **peripheral payloads** in a distrbuted manner. AMQP also allows Telemetry to have interoperability with many programming languages, allowing for **peripheral processors** to be language agnostic.

RabbitMQ is an open source, industry standard AMQP broker.

9.4 Application Layer: JSON

JSON is a human-readable, object encoding format. It is an industry standard format utilized in web applications.

9.5 Controller: Go

Go is statically typed, memory safe programming language. Go was chosen for the following features:

1. Type checking.
2. Memory safety.
3. Low-overhead runtime.
4. High performance networking.
5. Concurrency via message passing.

Utilizing Go, Telemetry will be lightweight, performant, and reliable.

10 Scenarios

10.1 New Peripheral Connected

The following will occur when a new peripheral is connected to a microcontroller.

1. User connects new **peripheral**.
2. **Microcontroller** detects new device, sends **registration frame**.
3. **Controller** receives **registration frame**, creates a new inactive device record in the **database**.
4. **Dashboard** will indicate a new peripheral is ready for activation.
5. To activate the **peripheral**, user will be prompted for:
   (a) Unique peripheral identifier (i.e. basementMoisture1)
   (b) Common identifier (i.e. Basement Moisture)
   (c) Polling interval (i.e. 30 seconds)
   (d) Peripheral type (i.e. adafruit_moisture)
6. After submitting the provisioning form, the **controller** will update the **database** entry for that device. The **controller** will also send a **provisioning frame** to the corresponding **microcontroller**, specifying the new schedule.
7. **Microcontroller** stores the new schedule in volatile memory (RAM).
8. When data from the **peripheral** becomes available, it will become selectable on the **dashboard** for data viewing.

10.2 User Changes Polling Interval

The following will occur when the user changes the polling interval of a peripheral.

1. User selects **peripheral** on **dashboard**, selects modify.
2. User enters a new polling interval (i.e. 10 seconds) and saves setting.
3. **Controller** updates database entry for the device. **Controller** also sends a new **provisioning frame** to the **microcontroller**, specifying a new schedule.
4. **Microcontroller** stores the new schedule in volatile memory (RAM).

10.3 User Deprovisions Peripheral

The following will occur when the user deprovisions a connected peripheral.

1. User selects **peripheral** on **dashboard**, selects delete.
2. User agrees to user confirmation, **peripheral** is removed from dashboard.

3. **Controller** deletes database entry for the device. **Controller** also sends a new **provisioning frame** to the **microcontroller**, specifying a new schedule.
4. **Microcontroller** stores the new schedule in volatile memory (RAM).

10.4 Normal Scheduling Operation

The following is the main event loop that occurs under normal operation.

10.4.1 Microcontroller

The event loop will be preempted if the **microcontroller** receives a **provisioning frame** or **request frame** sent from the **controller**.

1. **Microcontroller** reads schedule from volatile memory (RAM).
2. **Microcontroller** will iterate over the list of provisioned **peripherals**:
   (a) If there exists a previous read for an I^2C address and a new reading is not required yet, skip the **peripheral**.
   (b) If there exists a previous read for an I^2C address and a new reading is required, or if a previous reading does not exist, perform collection transaction. After the raw **peripheral payload** is collected, publish the data over MQTT in a **payload frame**.
3. Repeat.

10.4.2 Controller

The controller will run this event loop while concurrently responding to HTTP requests for the **dashboard** and **API** service.

1. **Controller** receives **payload frame** from **microcontroller**, containing raw **peripheral payload**.
2. **Controller** publishes raw **peripheral payload** over AMQP.
3. **Processor** consumes raw **peripheral payload** over AMQP. **Processor** parses raw I^2C data into a list of JSON-encoded key-values (**peripheral data**). **Processor** publishes **peripheral data** over AMQP.
4. **Controller** consumes JSON-encoded **peripheral data** from AMQP. **Controller** inserts data into database and publishes data over websocket.

Software Design Description

Code Guide

Preamble

Emily Horsman and Tanner Ryan are responsible for this aspect of the project.

Hardware/Software Requirements

Language Summary: C++, Go, Haskell, JavaScript, Python

Hardware

1. An ESP32 or other Arduino-compatible microcontroller. We used a HUZZAH32 from Adafruit during development. `https://www.adafruit.com/product/3405`
2. I^2C based sensors. We used an SHT31, LIS3DH, TCS34725, and AMG8833 during development to ensure our protocol works on a wide variety of sensors.

Microcontroller Firmware

1. C++11. This is the primary language the firmware is written in.
2. PlatformIO. This is a development toolchain to target the embedded hardware. `https://platformio.org`
3. nanopb v0.3.9.5. An implementation of Protocol Buffers designed for embedded systems. Included in this repository under `prototypes/lib/Nanopb`. `https://jpa.kapsi.fi/nanopb/`
4. MQTT Arduino Client. `https://github.com/knolleary/pubsubclient/`
5. Python 3.6. There is some rudimentary templating performed at compile-time for the provisioning web server running on the microcontroller. `https://www.python.org`

Protocol and Networking

1. `protoc` to compile and test Protocol Buffer schemas/messages. `https://developers.google.com/protocol-buffers`
2. VerneMQ. This is an MQTT broker. `https://vernemq.com`
3. RabbitMQ. This is an AMQP broker. `https://www.rabbitmq.com`
4. Docker. The backend, database, web server, and brokers are all provisioned with a Docker configuration file: `backend/Dockerfile`. `https://www.docker.com`

Backend

1. Go v1.14.2. This is the primary language the backend is written in. `https://golang.org`
2. Caddy v2. This is a web server which fronts the Go application server. `https://caddyserver.com`
3. PostgreSQL v12. Data from micro controllers is logged to a Postgres database. `https://www.postgresql.org`

Peripheral Processors

1. Haskell 2010. Peripheral processors can be written in any language with an AMQP library. This repo provides an example processor writte in Haskell (`sht31-controller/`) and includes a support library for other processors written in Haskell. `https://www.haskell.org`
2. Cabal to install Haskell dependencies in a standard way. `https://www.haskell.org/cabal/`
3. Aeson v1.4. `https://hackage.haskell.org/package/aeson`
4. base64-bytestring v1.1. `https://hackage.haskell.org/package/base64-bytestring`
5. AMQP client v0.19. `https://hackage.haskell.org/package/amqp`

Dashboard

1. JavaScript following modern ECMA standards is the primary language the dashboard is written in (along with HTML/CSS). One only needs a modern evergreen browser.
2. `npm` to install JavaScript packages. `https://www.npmjs.com`
3. React v16. `https://reactjs.org`
4. React Victory v34. `https://formidable.com/open-source/victory/`

Completed Features and Details

1. An abstraction for generalizing I^2C sensors. This is reflected in the architecture of the entire repository. A primary artifact with detailed comments can be found in `protocol/telemetry.proto`. This is a Protocol Buffers definition for messages that generalize the behaviour of I^2C sensors. A processor such as the one found in `sht31-controller/` can then specify the interaction with a sensor in a declarative fashion — without writing new application logic or physically updating firmware. The low-level task of reading from the bus and understanding the sensors' protocol is decoupled from processing the data.
2. A REST API for retrieving logged data from sensors and connectivity information. Includes various filters such as time ranges. `backend/controller/api/api.go`

3. A WebSocket API to allow web browsers to receive data in real-time without polling an API. `backend/controller/api/websocket.go`
4. A database for logging retrieved data and microcontroller provisioning. `backend/controller/db/init.go`
5. Docker configuration to provision the web server, database, and brokers. `backend/Dockerfile`
6. A sample processor written in Haskell to decouple application logic involved in using I^2C peripherapls. This includes a support library for other processors written in Haskell to use. `sht31-controller/`
7. A sample web application which uses WebSockets to display data from a sensor in real-time. `dashboard/src/App.js`
8. Automatic discovery of I^2C sensors when they are physically connected or disconnected. `prototypes/src/I2CManager.cpp`
9. A generic non-blocking scheduler using modern C++11. `prototypes/src/Scheduler.cpp`
10. A web application served by the microcontroller to allow a user to configure data without changing the firmware. This data persists on reboot. `prototypes/src/WiFiProvisioning.cpp`
11. A runtime on the microcontroller for generalizing I^2C behaviour with declarative configurations. `prototypes/src/I2CRuntime.cpp`

A Tour of the Simplest Case

Below is a description of the components involved in the simplest task of Telemetry: reading data from a sensor connected to a microcontroller, storing it in a database, and serving it through a REST and WebSockets API. The primary feature of Telemetry is the architecture that allows I^2C peripherals to be generalized and used without custom application logic. The following tour is only showing one path of what this architecture allows. It is a limited view that aims to highlight aspects of the codebase and how it works together — not the entire scope of the project.

1. A microcontroller running the Telemetry firmware is powered on.
2. It broadcasts a WiFi access point and serves a web application. This allows users to provision the microcontroller with network connection information without needing to hard-code values in firmware and have physical access to the microcontroller. `prototypes/src/WiFiProvisioning.cpp`
3. A user provisions the microcontroller and the microcontroller connects to the given WiFi access point.
4. A sensor is plugged into the microcontroller. The sensor is automatically discovered. The Telemetry firmware consists of an event loop and state machine which poll the I^2C bus for peripherals connecting and disconnecting — `prototypes/src/I2CManager.cpp`. It also manages the firmware's multiple tasks by loosely emulating threads — the microcontroller is not running an operating system that gives threads. We wrote a generic scheduler: `prototypes/src/Scheduler.cpp`.
5. A message is sent to the backend through the MQTT broker describing the connected sensor. `prototypes/src/MQTTManager.cpp`, `prototypes/src/TelemetryProtocol.cpp`
6. The backend asks the online processors if any are responsible for the described sensor. TODO

7. The responsible processor sends a declarative configuration for the sensor to the AMQP broker.
8. The backend delivers this to the appropriate microcontroller through the MQTT broker.
9. The firmware's runtime stores this configuration and will use it to poll data from the sensor. `prototypes/src/I2CRuntime.cpp`
10. The microcontroller reads data from the I^2C bus and sends the raw bytes to the backend through MQTT.
11. The backend proliferates this to the appropriate processor through AMQP.
12. The processor converts the raw bytes to semantically correct data (such as a temperature value). `sht31-controller/Main.hs`
13. The processor sends this data to the backend through AMQP. `sht31-controller/Support.hs`
14. The backend writes this data to the database. `backend/controller/db/`
15. The backend pushes this new data to any open WebSocket connections. `backend/controller/api/websocket.go`
16. A dashboard with an open WebSocket connection reads this message and graphs the data over time. `dashboard/src/App.js`

Statistics

Using the tool `sloc` to quantify 'significant' lines of code:

```
> sloc --exclude 'binary|docs|.*\.pio.*|.*vendor.*|Nanopb|.*node_modules.*' .

---------- Result ------------

  Physical :  5674
    Source :  4140
   Comment :  974
```