Telemetry

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

April 2020

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.
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

[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.

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.

Software Requirements Specification

Software Design Description

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

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.

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^{2}C 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^{2}C 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^{2}C 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
```