Module 1: Software

The software module consists of two submodules: **The Protocol** and **The Bridge Driver**.

The Protocol

A single physical device can represent several nodes on the network, each fufilling a given role:

1. **Sensor Device**: Collects information (often biometric) and communicates it to a control device.
   1. Can be set to regularly send out information at a negotiated sampling rate.
   2. Can be set to notify about specific events (i.e. heart rate metric falling below a threshold).
2. **Delivery Device**: Delivers an output that may effect broader system state. Can accept input from a control device.
3. **Control Device**: Processes sensor data and can make requests to delivery devices to change their system parametes (e.g. delivery rate).

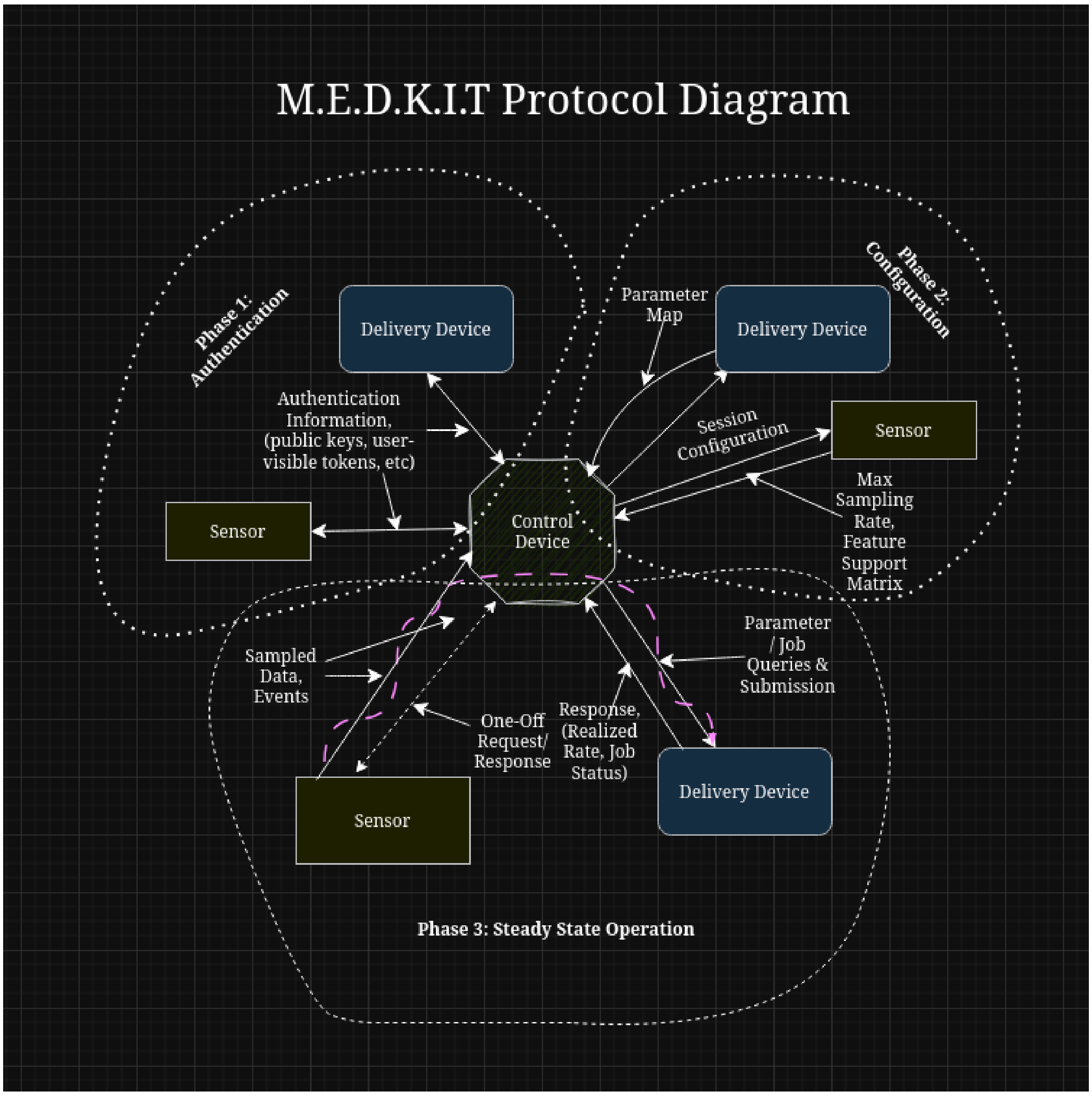
To use the example of a diabetes care network, a typical system could consist of a glucose monitor acting as a **Sensor**, communicating with a closed-loop integration device functioning as a **Control**, which modifies the insulin rate of a pump, which acts as a **Delivery** device.

This setup is very rigid in the relationship between physical device and role, but one has to look no further than their own pocket to find a potential device that distrupts this pattern: the modern smartphone.

Take for example a system in which a user downloads an app on their phone which allows them to track their diet, input expected excercise plans, and optionally connect a glucose monitor and uses that to calculate appropriate dosage rates for an insulin pump. In such a system, a single physical device can act as a number of sensors, while also behaving as a control device. Many insulin pumps may have a glucose level display that would require querying a sensor directly.

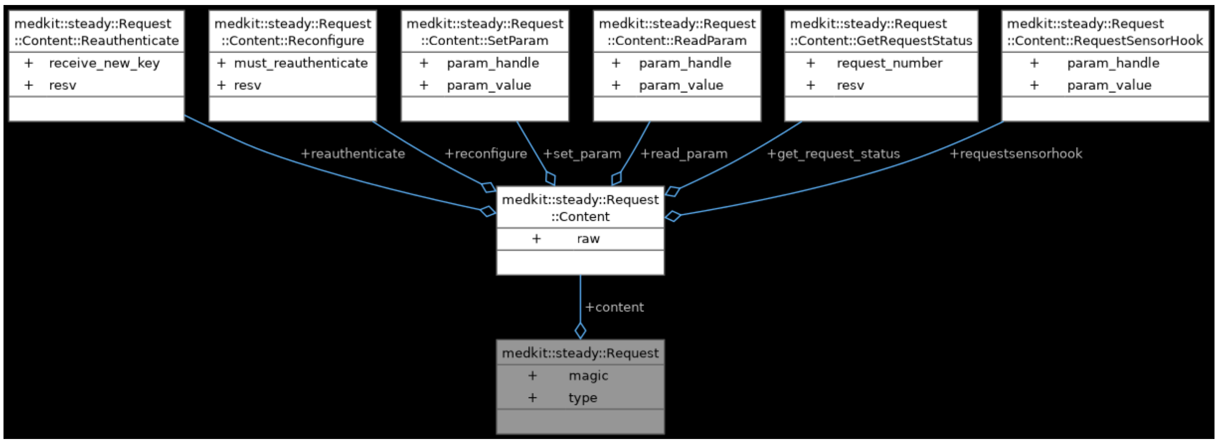
To solve the issue of nodes wearing many hats, the protocol was designed around the individual relationships between nodes. Sensors and delivery devices can seek out multiple control devices, first authenticating themselves. After "authentication" is complete, the individual parameters for each side of the relationship is set by negotionation in a process known as "configuration". After configuration, the types of messages in the protocol are limited in scope to simple request-response dichotomies.

Due to practical and time constraints, the majority of the implementation was focused on the "steady-state" portion, and more complete analysis must be done before an authentication protocol is complete.


M.E.D.K.I.T Protocol Diagram
Phase 1: Authentication
Phase 2: Configuration
Phase 3: Steady State Operation
Delivery Device
Delivery Device
Delivery Device
Sensor
Sensor
Sensor
Control Device
Authentication Information, (public keys, user-visible tokens, etc)
Parameter Map
Session Configuration
Max Sampling Rate, Feature Support Matrix
Sampled Data, Events
One-Off Request/ Response
Response, (Realized Rate, Job Status)
Parameter / Job Queries & Submission

In order to make use of the protocol easily accessible, a reference implementation was provided in the form of a header-only C++ library. This choice was made due to the widespread use of C in embedded programming, with its unions, structs, and packed bit fields allowing unambiguous speicfication of data ordering.

For an example of this leveraging, examine the type diagram for the Steady State phases's Request packet, which allows a single very-small (16 byte) network packet to represent various messages, from the renegotiation of a new configuration, to notification of an event, or a change in delivery policy.


medkit::steady::Request ::Content::Reauthenticate
+ receive_new_key
+ resv
medkit::steady::Request ::Content::Reconfigure
+ must_reauthenticate
+ resv
medkit::steady::Request ::Content::SetParam
+ param_handle
+ param_value
medkit::steady::Request ::Content::ReadParam
+ param_handle
+ param_value
medkit::steady::Request ::Content::GetRequestStatus
+ request_number
+ resv
medkit::steady::Request ::Content::RequestSensorHook
+ param_handle
+ param_value
+reauthenticate
+reconfigure
+set_param
+read_param
+get_request_status
+requestsensorhook
medkit::steady::Request ::Content
+ raw
+content
medkit::steady::Request
+ magic
+ type

The template metaprogramming technique of "traits" was employed in order to abstract away networking interfaces and allow the reference implementation to not be directly tied to any specific networking hardware or library.

The Bridge Driver

The bridge driver is the code that is initially present in the bridge device, which allows for remote reconfiguration and hot reloading of code.

Due to the only partial implementation of the configuration stage in the library code, step 1 is accomplished through directly programming the EEPROM of the device over UART, instead of taking advantage of the hot reloading capacity of the protocol:

```
0. Authenticate to "programming control device" (not done in current demo due to lack
of authentication protocol)
1. Obtain configuration for host device (pre-programmed in current demo)
                    |
                    v
2. Observe host device outputs :-> Communicate if configuration dictates
3. Observe network inputs :-> Translate to communication with host device if
configuration dictates
4. Repeat step 2.
```

Appendix Software Listing

Due to the length of the software listing and the number of files, the entirety of the code, as well as the original final presentation and software documentation is available in the following git repository:

- https://github.com/connellpaxton/medkit