Requirements:

Data Generator:
- We want to generate sensor data to be sent and stored.
- We should be able to generate random data or use a source CSV file

Object System:
- We want to make a generic system that can emit & receive data from multiple objects (e.g. car + toaster)
- Objects have a set of sensors with different data types and different send frequencies

Sensor System:
- We want to dynamically download and cache sensor information for a given object.
- The system will only download ~~new~~ sensor information.
- Each sensor will contain relevant data type and frequency info, along with identifiers.
- Additionally, sensors will come w/ transformation data that will let us convert raw voltage to an appropriate unit.

Channel System:
- Data for a given object will be sent through a channel. For example, a car will receive sensor data over CAN. Sensors will have a "channel decoding ID" that will help them get the appropriate data.

Transceiver System:
- We will have a generic UDP/TCP system that will give an object an interface to start telemetry sessions, receive sensor information, and receive messages (e.g. display a remote message on a driver's dash). The interface will also let objects send sensor data.

VFDCP:
- "Variable Frequency Data Compression Protocol".
- We will use UDP to send data at a variable frequency in a compressed format. Additionally, we will only send data if it has changed from it's previous value by some ϵ scaled by the sensor's min and max value.

Storage Sys.:
- We want to store (flash in HW terminology) sensor information so we don't waste "RTE" data fetching.
- We also want to store a local copy of a session's telemetry data for a "source of truth" as sending over UDP will cause loss, we can fetch full data from the HW.

VFDCP:

Definitions:

We have a set of sensors $S_1, S_2, \ldots, S_n$

Each sensor S_i has:
- A frequency f_i
- An id s_i
- A max value M_i
- A min value m_i

The function to determine a "substantial change" in the present value a sensor worthy of transmission is:

```
f(currentValue, prevValue, min, max, ε):
  range = the absolute value of the range between min, max
  minChange = range * ε
  if(|currentValue - prevValue| > minChange)
    worth sending
  else not worth sending
```

A "frame" is a set of sensor data that will be sent at a given timestep.

The frame will have the following structure:

```
{
  sensor_ids;
  sensor_values;
}
```

The sensor_ids are an ordered list of small identifiers that allow us to decode an ordered list of unevenly sized sensor values inside sensor values.

The frame will only contain sensor values if the sensor's frequency lines up with the given timestep AND the sensor's value has significantly changed.

VFDCP Goals:
- Reduce cost of data transmission by only sending data that we need.
- Tradeoff reliability for speed of packets.

We decode VFDCP frames with a copy of the sensor list on the receiver.

VFDCP tradeoffs and potential problems:

- Complicated...
- Reliability - Sending over UDP will cause loss and potentially out of order data.
- Generic typing - all in C will make this tricky, but for the simulator, we have C++, should try to use C syntax.

VFDCP benefits:

- Huge cost savings in the average case.
- Efficient.

Let's take the SR-21 FSAE vehicle for example:

40+ sensors ($S_1, S_2, \ldots, S_{90}$), each with a different freq. f_i, each with different value sizes (double, float, byte).

Without VFDCP:

- We have a max frequency $f_{max} = 60$
- We have a min frequency $f_{min} = 1$
- We send data at 60 Hz with the most recent value of each sensor
- Our data per second is:

Average

$$\left(60\tfrac{1}{s} \times 8 \text{ bytes}\right) + \left(60\tfrac{1}{s} \times 40 \text{ sensors} \times 4 \text{ bytes/sensor}\right)$$

UDP header — Amount of sensor data sent

$$= 480 \text{ Bps} + 9{,}600 \text{ Bps} = 10{,}080 \text{ Bps}$$

Our average session is 20 minutes:

$$= 10{,}080 \text{ Bps} \times 60 \text{ s/min} \times 20 \text{ min} = 12{,}096{,}000 \text{ bps}$$
$$= 11.54 \text{ MB}$$

We have about 200 hours of testing per season (600 sessions):

$$\approxeq 6.76 \text{ GB}$$

Our cost per MB is ~$0.03:

Cost per season = $207.72 → Yikes! We don't have this.

There was extra overhead for identifying sensors, which would bring the total to ~$300. (Extra data sent specifying sensor:value).

With VFDCP:

We have an average frequency of 5 Hz.

Many sensor values don't change often, let's assume, that on average, only 75% of sensor values need to be sent at their respective time range.

Doing the above calculations again:

Data per second:

$(60\frac{1}{s} \times 8 \text{ bytes}) + (60\frac{1}{s} \times 5 \text{ sensors} \times 4 \text{ bytes/sensor})$ ← Average number of values sent

UDP header

constraint

$+ (60\frac{1}{s} \times 10 \text{ bytes}) \times 0.75$ ← Sensor IDs (decode information) ↓ 1 byte per sensor (255 possible sensors)

$= (480 \text{ Bps} + 1{,}200 \text{ Bps} + 600 \text{ Bps}) \cdot \frac{3}{4} = 1710 \text{ Bps}$

Data per season:

$\cong 1.14$ GB

Cost per season:

$\cong \$35.22$ → Not bad, still expensive ↗ Nice!

Net improvement = $1 - (\$46.97/\$300) \approx 89\%$

One more optimization...

We need to deal with byte padding.

```
struct x {
  char ci
  int  bi
};
```

This struct has 5 bytes of data, but 8 bytes of allocated memory! This is because data is stored in words (4 bytes).

Padding

The structure in memory looks like cXXXiiii.

We have wasted data! How do we solve this?

Big endian

Approach 1:

We have a constraint of a library that destructures structs, not bytes in Python. Meaning we need to view data in words.

Our receiver can currently destructure any data inside words.

For example, say we have a = int, b = short, c = char

Best case: aaaabbcX

Worst case: bbXXaaaacXXX

Our receiver can handle both cases.

Intuitively, we can minimize padding by ordering data largest → smallest. We have a solution, at maximum, we will have 3 bytes of padding. Nice!

Approach 2:

We can send the exact number of bytes our data fits into by using a byte array. We will have to create our own byte parser in the python server.

This might make the C/C++ code easier though.

Is it worth the extra effort to save 3 bytes? Maybe at scale, but the dev time is not worth it right now.

Design

Primary Systems:
- Data Generator
- Object/Sensor System
- Transceiver System
- Channel System
- VFDCP System
- Storage System

System breakdown:

Data Generator:
- Sensor sync (need to know which sensors to generate data for)
- Random data generation
- Data gen from file
- Publishing to a channel

Channel System:
- Maintain current & previous values for sensors
- Allow subscription to particular sensor data
- Allow read for a particular sensor using an id

Object System:
- An object interacts with sensors, channels, and the transceiver and acts as a mediator.
- Templates to support an object like a car

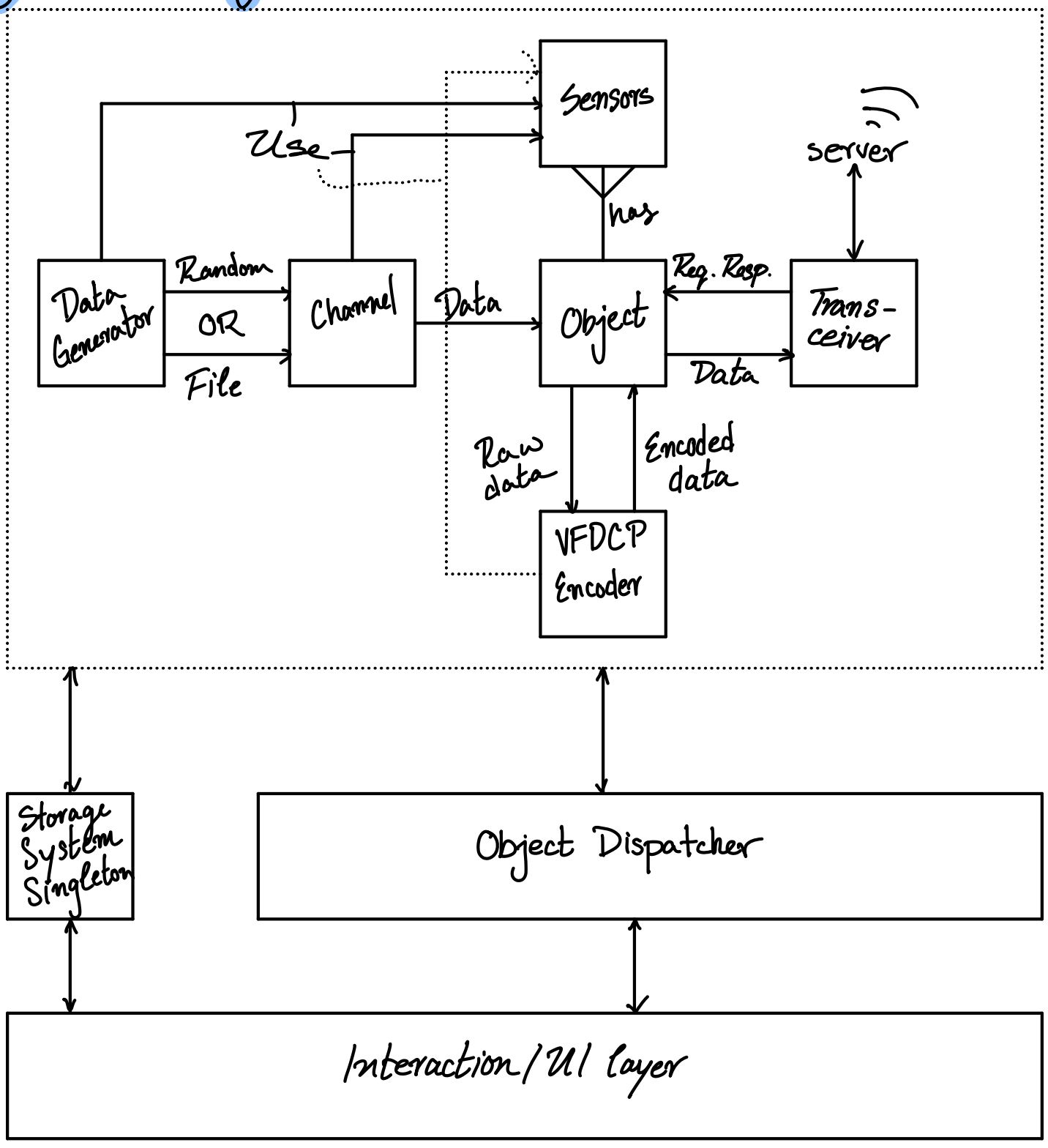
Sensor System:
- Objects retrieve their sensors through the transceiver and propagate them where needed
- Effectively stores and manages sensors.
- Reconciles updated sensors
- Stores sensor information locally using storage system

Storage System:
- Stores sensor information for a particular object via serial number
- Stores session data (object telemetry turned on then off) in csv format.

Transceiver System:
- Receives/requests server data, including starting a session, asking for sensors/sensor diff, and other messages.
- Transmits sensor data with VFDCP over UDP

VFDCP System:
- Acts as an encoder by receiving current and previous sensor information, along w/ all sensor information.
- Return encoded byte array to be sent out.

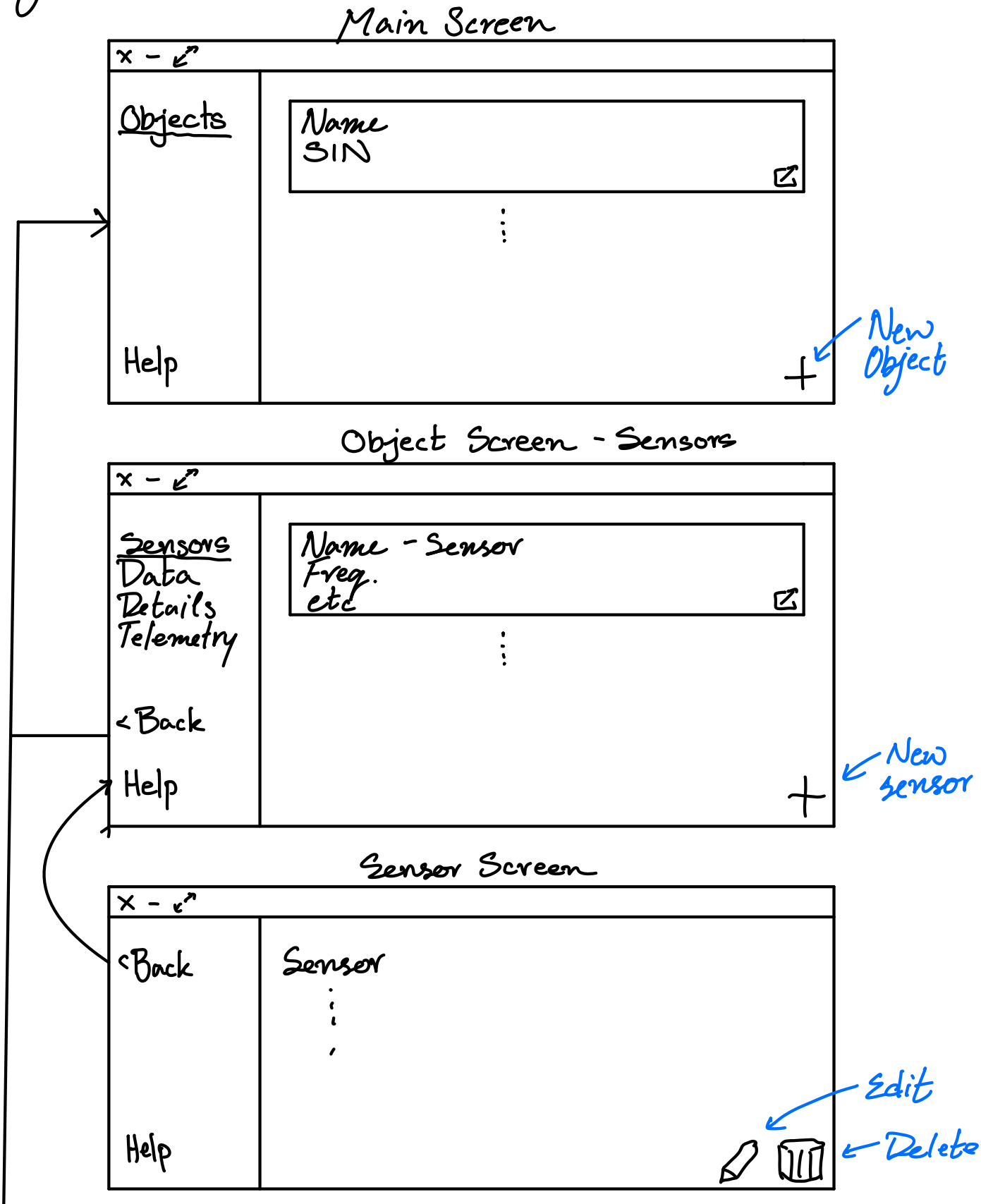
High-level Design

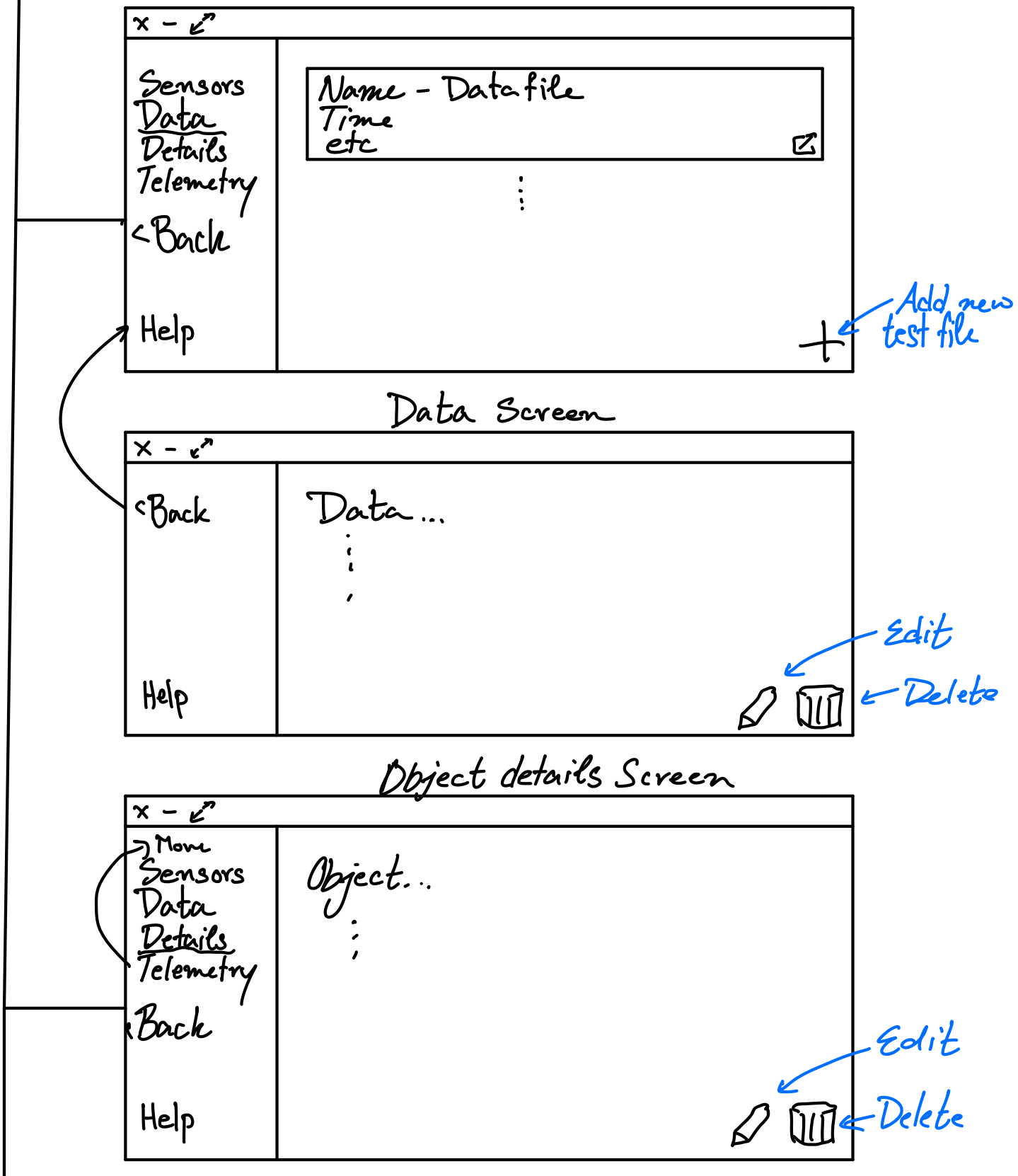


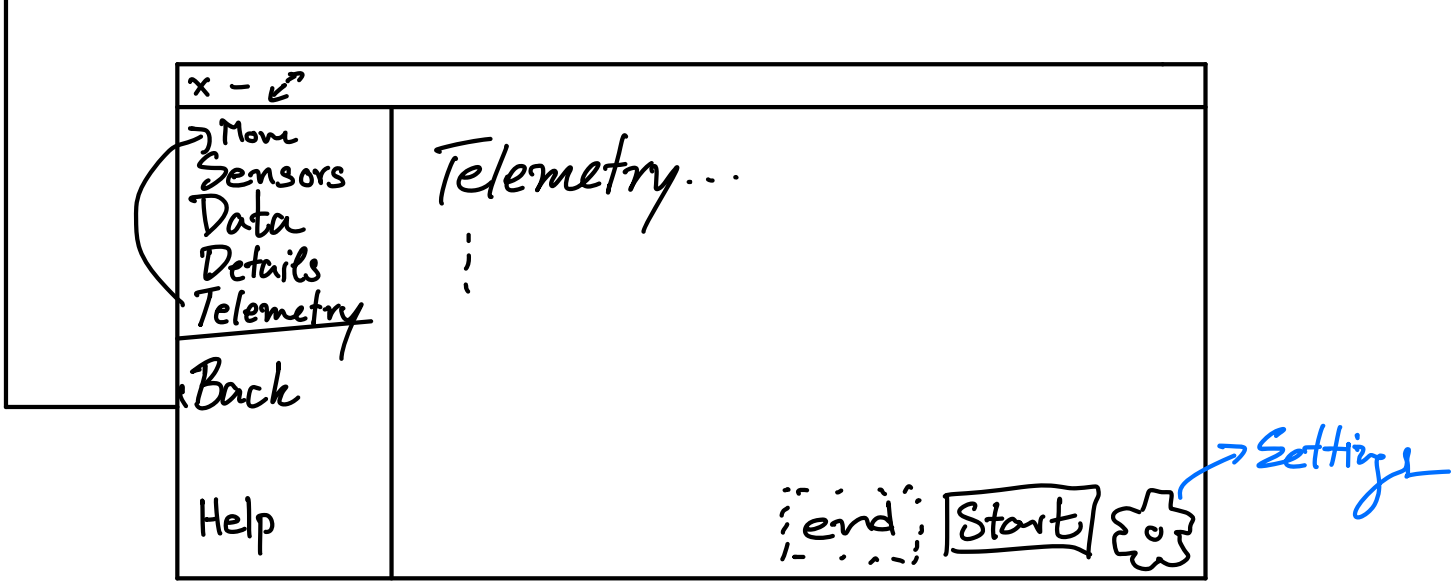
Interaction / UI Layer

We want to interact with the system in an easy to use matter. We also want to run multiple objects simultaneously, the system as a whole is a multiobject telemetry simulator.

Using Qt, we can make some UI:




x - ↙
Sensors
Data
Details
Telemetry
<Back
Help
Name - Datafile
Time
etc
Add new test file
Data Screen
x - ↙
<Back
Help
Data...
Edit
Delete
Object details Screen
x - ↙
Move
Sensors
Data
Details
Telemetry
Back
Help
Object...
Edit
Delete



UI is meh, proper mockups should be made. Needs to be easy and convenient to use.

I realized we need to fetch objects from our API, we will need a higher level receiver to get top level data.

In the end, we don't want to store anything locally, all should be stored on the server. For testing purposes, we will have local stores of data.

Plan

Send it!