CSU33031
Assignment 1 Report

Lydia MacBride
19333944

October 2021

Contents

1 Introduction

All code is available at my GitHub repo for this project at: `https://github.com/LydiaUwU/CSU33031-Assignment-1`

This document details my approach, design and implementation for CSU33031 Assignment 1. This document is as thorough as I could make it and is therefore very, very information heavy. Please feel free to skim through sections as you deem necessary.

This implementation and protocol design aims to satisfy the following criteria.

1.1 Criteria

- Design a protocol that is sent via UDP packets that contains its own header design to support the required functionality.
- Design the protocol to handle a number of sensor devices and allow them to publish their readings to a central broker device.
- Design dummy sensor devices that publish placeholder data to the broker at regular intervals, mimicking a real sensor device (i.e. thermometer).
- Design a dashboard device that can subscribe to specific device groups using our protocol.
- Design a broker device that can manage incoming data from various devices and store data or forward packets to other devices when necessary.
- Design dummy actuator devices that can be set via publication commands to specific values (i.e. air conditioning units).
- Extend the dashboard to allow a user to publish commands to various device groups.
- Extend the broker to forward publication commands to all devices within a group specified by the user on a dashboard device.
- Clearly document the designs of the protocol and implementation.

2 Dashboard Command Reference

2.1 sync

Running this command sends the `syn` packet type to the broker device, which will in turn begin the device synchronisation process. When this process is complete the dashboard device will have an up to date record of the devices connected to the network.

Note: The functionality of this command also occurs after a handful of other frequent operations so there is not much need to manually run this command unless you need an the most recent information from the broker.

2.2 ls

Running this lists the devices on the network that are currently known to the dashboard.

Arguments

-sensors

Running `ls -sensors` will list only the known sensor devices on the network.

-actuators

Running `ls -actuators` will list only the known actuator devices on the network.

2.3 sub

This is the **Device/Group Subscription** command.

Running this allows a user to subscribe the dashboard device to a device group or a specific device given its ID.

Arguments

<group>

Running `sub <group>` where `<group>` is the name of the device group the user wishes to subscribe to.

-i <type>:<id>

Running `sub -i <type>:<id>` where `<type>:<id>` is the type and device ID of the device the user wishes to subscribe to.

2.4 pub

Running this allows the user to set the value of any given actuator or actuator group on the network.

Arguments

<group>

Running `pub <group>` where `<group>` is the name of the device group the user wishes to publish the command to.

-i <type>:<id>

Running `pub -i <type>:<id>` where `<type>:<id>` is the type and device ID of the device the user wishes to publish the command to.

2.5 debug

Running this command toggled the `debug` value in `client-main.py`. When this value is **True** all the `print_d()` calls in the threads that arent `ProcessInput()` will display their output.

Warning: This debug output is messy and not designed for the end user to interact with, also due to the nature of multi-threading, there is a fair chance that one of the other threads will output as you are typing a command and interrupt you. You can always disable debug output by running the command a second time.

2.6 exit

This command ends and joins the user input processing thread, `ProcessInput()`, and the packet sending thread, `SendPacket()`, and then exits the `client-main.py` script, allowing for a clean shutdown of the dashboard device.

2.7 help

This command prints the the list of available commands, their arguments and a brief description of each.

3 Design Decisions

3.1 Overview

I decided to do my implementation using *Python*, mostly because I like the syntax and it offered a number of libraries that proved useful to my implementation. Of note are the `socket` and `threading` libraries. The `socket` library allowed me to initialise a UDP socket and send/receive packets between devices with relative ease. The `threading` library proved to be very useful for managing flow control and reducing the possibility of missing packets as it allowed me to run my message sending and reception processes in separate threads, removing an issue I was having where incoming packets were missed while the device was processing something else.

It is worth noting that *Python* generally tends to have more performance overhead compared to other languages due to it being interpreted rather than compiled, but given its ease of use and the limited scope of this project any performance loss due to this was not a significant concern. That said, I designed the protocol in such a way that it should be readily implementable in any given language, and it should even support a system where each device is operating on a different language to each other.

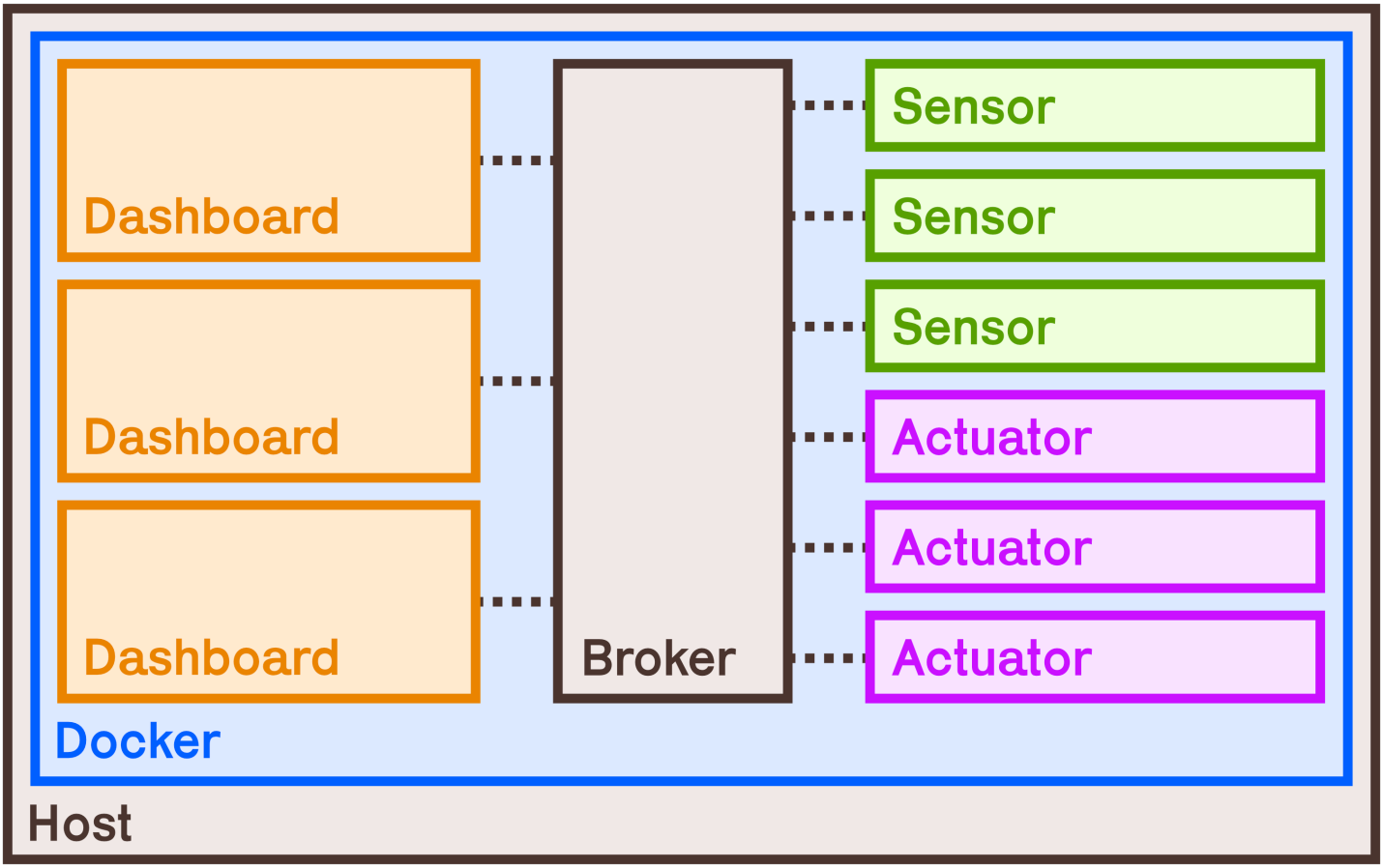
For my own ease, I decided to build my implementation to utilise *Docker*, as I felt it had enough features relative to its complexity. I was considering using *Kubernetes* during my planning stages but figured it was more complex than necessary for this design.

3.2 Protocol

For the protocol, given that this was an assignment expected to be ready demonstrable, I prioritised human readability. Using this principal I decided that there should be a number of different packet types, each with a uniquely identifiable three character long name, and each packet would contain a number of fields for device types, IDs and data all separated by semicolons and each packet would be encoded by the `UTF-8` standard. With this structure I was able to use *Python's* string parsing features to easily create and send packets, and then split them into their various data sections with ease, without having to worry about specific lengths or indexes to split each section. This of course comes at the expense of packet size, encoding each packet as a string rather than as few bits as possible, however I felt human readability to be more important.

As UDP does not have any kind of acknowledgement procedure built-in, I used the `ack` packet type to send confirmation of package reception from the receiver back to the original sender. For more information on how my implementation of acknowledgements work see the **flow control** section of this document.

3.3 Network Topology



Using *Docker* I implemented the above network topology. For demonstration purposes there are only three dashboards, sensors and actuators but my design can accommodate an arbitrary amount of each. I believe that my implementation would be able to run on physical devices on a network rather than a container with a small amount of modification, mostly to do with how the broker's address is resolved as it presently uses *Docker's* native DNS resolution.

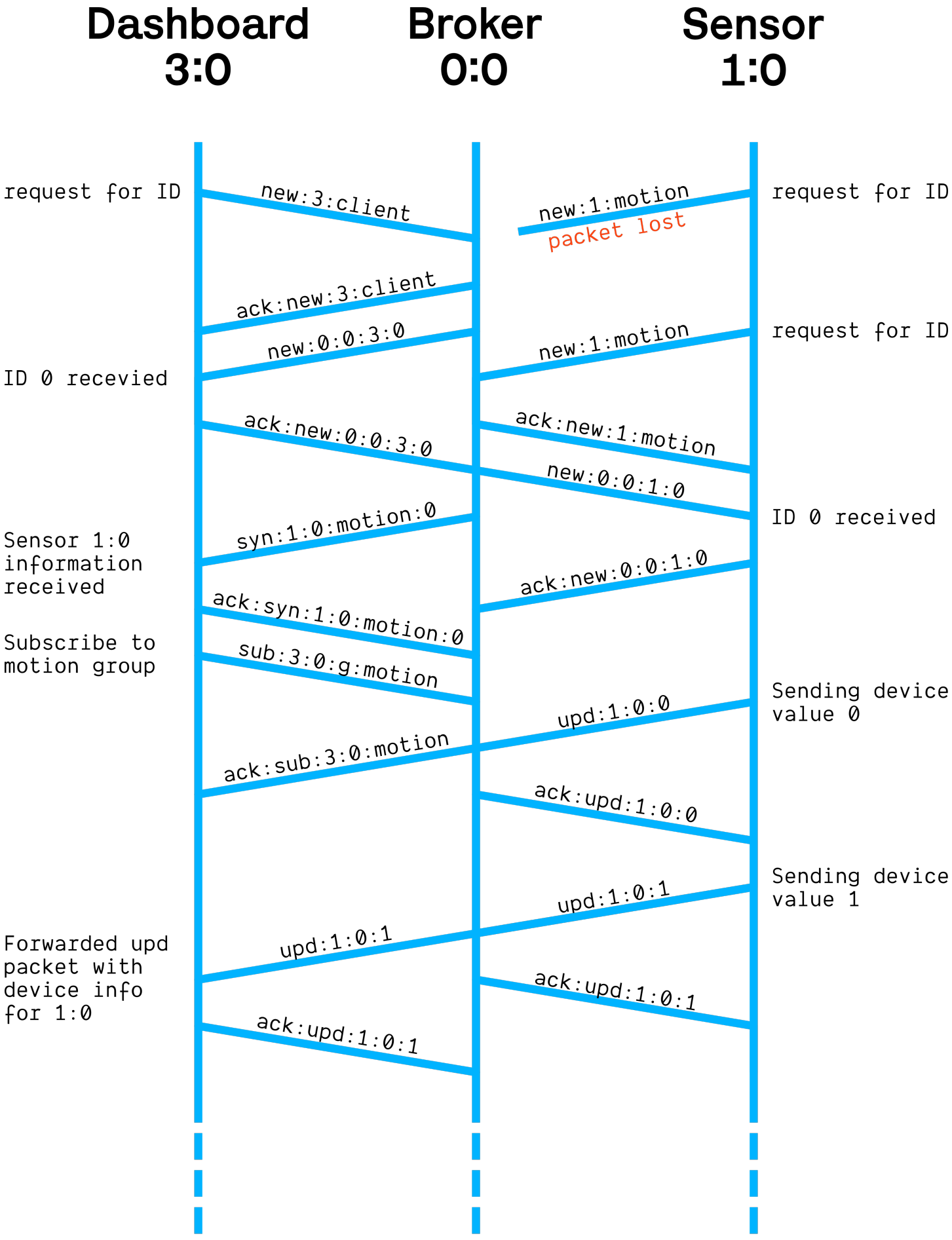
4 Protocol Overview

4.1 Flow Control

Integral to the smooth operation of this protocol is the number of flow control features implemented both in the protocol and in each device's implementation, most critically being the `ack` packet type and the packet queues in each device. For every packet a device wishes to send, it appends it to its packet queue. In the implementation the queue is a *Python* `list()`, which outgoing packets are appended to as necessary. The `SendPackets` thread on each device then cycles through all the packets in the queue and sends them to their specified address. This plays two important roles, firstly it allows us to move packet sending to its own thread allowing for more continual sending of packets, and secondly it allows us to maintain a list of packets that weren't successfully received by the destination device. This is due to packets remaining in the queue until the expected `ack` packet is received, only then is it removed from the queue. This means that packets will continue to be sent until they are successfully received and acknowledged, guaranteeing reliable data transfer between devices.

In the event the packet is received by the target device but the acknowledgment is lost the original packet is resent until the acknowledgement is successful received. The repeated transfer of packets is acceptable for the purposes of this protocol as it is designed to deal with data where time sensitivity isn't the most critical concern and each device is equipped to handle duplicate packets without issue. In a future revision of this protocol, we could ensure that each device only handles the most recent packets from devices by the inclusion of a timestamp field in the protocol, however for demonstration purposes I felt that this feature was not currently necessary.

The flow diagram below demonstrates the exchange of packets in a simple three device network containing a dashboard, the broker and a sensor. The diagram includes the device initialisation process for the dashboard and sensor, where each device requests a unique ID (remember, IDs are assigned on a device type basis), the new sensor device is synced with the dashboard, the dashboard subscribed to the device group `motion` and the broker receives and forwards an `upd` packet to the dashboard. We also see a demonstration of what happens when a packet is unsuccessfully transmitted, with the sensor's `new` packet failing to transmit, and then re-sending shortly after.


Dashboard
3:0
Broker
0:0
Sensor
1:0
request for ID
new:3:client
new:1:motion
packet lost
request for ID
ack:new:3:client
new:0:0:3:0
new:1:motion
request for ID
ID 0 recevied
ack:new:0:0:3:0
ack:new:1:motion
new:0:0:1:0
ID 0 received
Sensor 1:0
information
received
syn:1:0:motion:0
ack:new:0:0:1:0
ack:syn:1:0:motion:0
Subscribe to
motion group
sub:3:0:g:motion
Sending device
value 0
upd:1:0:0
ack:sub:3:0:motion
ack:upd:1:0:0
Sending device
value 1
upd:1:0:1
Forwarded upd
packet with
device info
for 1:0
upd:1:0:1
ack:upd:1:0:1
ack:upd:1:0:1

4.2 Packet Structure

Packets are generally structured in the following format:

`pck_type:src_type:src_id:dst_type:dst_id:data`

Where `pck_type` is the packet type, `src_type:src_id` is the sender's type and ID, `dst_type:dst_id` is the destination's type and ID, and `data` is the data of the packet such as a temperature value or a string.

4.3 Packet Types

4.3.1 ack

This is an **Acknowledgement** type packet.

Each device sends this packet type immediately after receiving a packet in the format `ack:<src_type>:<src_id>:<packet>`, where `<packet>` is the entire packet that was just received, including its type indicator. It is important to note that an `ack` packet should never be sent in response to a received `ack` packet as this will cause an infinite loop of acknowledgments being sent between two devices, effectively rendering them useless.

These packets play an important role in allowing a device to know when a packet it has sent has been properly received by its destination, as the sending device will keep its original packet in its queue and resend it if it does not receive a corresponding `ack` packet, only removing the original packet from the queue when this `ack` packet is received.

Example: `ack:new:0:0:3:0`, this is an acknowledgement for a new packet type sent to the broker from the client after receiving a new packet containing a device ID number.

4.3.2 new

This a **New Device** type packet.

This packet has two forms, one sent by non-broker devices during initialisation, and one sent by the broker in response to this packet from initialising non-broker devices.

Non-Broker Form

Sent in the form `new:<device_type>:<device_group>`, the broker then uses this to create a new device class and append it to the corresponding device list.

Example: `new:3`, this is a request for a new device ID sent from a dashboard device.

Broker Form

Sent in the form `new:<src_type>:<src_id>:<dst_type>:<dst_id>`. Of note is the value for `<dst_id>` which is the new unique device ID for the non-broker generated by the index the new device class was created and stored to in its corresponding device type list, that is, if there the new device is the 5th of that type that was instantiated, the ID value will be `4` as we start from ID `0`.

Example: `new:3:0`, this is a new device ID assignment sent from the server to a new dashboard device, assigning it the ID of `0`.

4.3.3 upd

This is a **Device Data Update** type packet.

This packet takes the form `upd:<src_type>:<src_id>:<data>`. Of note is the `<data>` value as this contains the relevant device data to be updated in the Broker's device lists and forwarded on to subscribed dashboard devices. As these packet types are always sent to the server the typical `<dst_type>:<dst_id>` fields are omitted.

Example: `upd:1:0:50`, this is an `upd` packet sent from a sensor device containing the `<data>` value of `50`, this value could correspond to an array of different units of measurment depending on the device group of the sensor that sent it, for example a sensor in the thermometer group would send values in degrees Celsius.

4.3.4 syn

This is a **Device List Sync** type packet.

This packet has two forms, one sent by dashboard devices and the other sent by the broker in response.

Dashboard Form

Sent in the form `syn:<src_type>:<src_id>`. This package is sent when the user enters the `sync` command and prompts the broker to send a broker form `syn` packet for each instantiated device.

Example: `syn:3:1`, this means that the dashboard device with ID `1` wishes to sync devices with the broker.

Broker Form

Sent in the form `syn:<dev_type>:<dev_id>:<dev_group>:<dev_data>`. This is sent for each device in each device type list containing the device's type, id, group and data respectively. The `<dev_data>` value is only sent if the requesting dashboard devices is subscribed to that device or its group, otherwise a value of `None` is sent.

Example: `syn:1:3:motion-detectors:idle`, this is `syn` type packet sent from the broker to a dashboard device that both requested to sync, and is subscribed to the sensor with the ID of `3`. As the dashboard is subscribed to the `motion-detectors` group, the device's data `idle` is sent to the dashboard.

4.3.5 sub

This is a **Dashboard Subscription Request** type packet.

This packet has two forms, one for subscribing to a device group and another for subscribing to individual devices.

Subscribing via Device Group

Sent in the form `sub:<src_type>:<src_id>:g:<group>`. Of note is the value `g`, this is used upon reception by the broker to indicate that this dashboard wishes to subscribe to a device group, further of note is the `<group>` value which indicated what group the dashboard wishes to subscribe to. This packet is sent when the user runs the `sub` command on a dashboard device.

Example: `sub:3:2:g:barometers`, this packet indicates that the dashboard with ID `2` wishes to subscribe to the `barometers` device group.

Subscribing via Device ID

Sent in the form `sub:<src_type>:<src_id>:<dst_type>:<dst_id>`. Of note are the vales `<dst_type>:<dst_id>` as they refer to the device type and ID the dashboard wishes to subscribe to. This packet is sent when the user runs the `sub` with the `-i` argument.

Example: `sub:3:5:2:8`, this packet indicates that the dashboard with ID `5` wishes to subscribe to the actuator with ID `8`

4.3.6 pub

This is a **Command Publication** type packet.

This packet has three forms, two sent by a dashboard for publishing via device group or ID and another sent by the broker to actuator devices.

Dashboard Publication via Device Group

Sent in the form `pub:<src_type>:<src_id>:g:<group>:<value>`. Of note are the following values: `g` which indicates that this is a publication via group type `pub` packet, `<group>` this is the device group that the dashboard device wishes to publish to and `<value>` which is the value the client wishes to set each actuator device to. This packet type is sent when the user runs the `pub` command on a dashboard device.

Example: `pub:3:1:g:fans:85`, this packet indicates that the dashboard device with ID `1` wishes to set all devices in the `fans` group to a value of `85`RPM (with the units being decided in the specific actuator device's own implementation).

Dashboard Publication via Device ID

Sent in the form

Broker Publication to Actuators

Sent in the form `pub:<src_type>:<src_id>:<dst_type>:<dst_id>:<value>`. Of note are the following values: `<dst_type>` and `<dst_id>` which denote the target device's type and ID, and `<value>` which is the value the client wishes to seat the target actuator device to. This packet type is sent when the user runs the `pub` command with the argument `-i` on a dashboard device.

Example: `pub:3:1:2:4:85`, this packet indicated that the dashboard device with ID `1` wishes to set the actuator with ID `4` to the value of `85`.

5 Implementation

5.1 Initialisation & Setup

Setup of the network is relatively simple using the provided shell scripts. To start clone the *GitHub* repo and enter it, then by running `./initialise.sh` the creation of all the required docker images and containers, alongside connecting them to a new network is handled for you. You can then run `./launch.sh` to launch all the required containers (note to launch them in interactive mode you must launch them individually). To launch just the client device run `./client.sh`. If you wish to remove the created containers run `./remove.sh`.

If you wish to create the containers and images manually each device type has its own sub directory containing all of its dependencies and the `Dockerfile` required to create them.

5.2 Components

5.2.1 Broker

The broker is the central device in the network as all devices communicate with the broker rather than communicating directly to each other. As the broker is central to all communications its core purpose is to store data from each sensor and actuator device on the network and to forward relevant packets/data to other devices on the network when it receives specific requests from the network.

Sent Packets

- `ack`: Sent in response to all received packets to acknowledge successful reception.
- `new`: Sent in the **Broker Form** upon reception of a **Non-Broker Form** `new` packet, the sent packet contains a new unique device ID which is generated using the current length of the relevant device type's list.
- `syn`: Sent in the **Broker Form** upon reception of a **Dashboard Form** `syn` packet. One packet of these packets is sent back to the dashboard for all sensor and actuator devices known to the broker.
- `pub`: Sent in the Broker Form upon reception either form of `pub` packet from a dashboard device. These packets are then sent to the specified actuators based on either the ID or group specified by the dashboard.

Received Packets

- `ack`: When received, the original packet the acknowledgement belongs to is removed from the devices queue.
- `new`: Received from initialising devices on the network, the broker will check if the device already exists in its list of known devices and will forward on either the existing or a new ID to the requesting device.
- `upd`: Received from all sensors and actuators on the network. The broker will store the new values from each incoming `upd` in the relevant device's information and forward the packets on to any clients subscribed to the sending device.
- `syn`: Received from dashboard devices, in response the broker will send a `syn` packet back for each known device to the network.
- `sub`: Received from dashboard devices, upon reception the broker will update the relevant clients list of device subscriptions.
- `pub`: Received from dashboard devices, upon reception the broker will forward the `pub` packet to all relevant actuator devices.

5.2.2 Dashboard

Dashboard devices are what the end user interacts with, providing a command line interface when launched (must be launched in interactive mode with `tty` enabled when using *Docker*, see **Setup & Initialisation** for more information). Once launched the user can use a variety of commands to interact with the dashboard itself and all other devices on the network, see **Dashboard Command Reference** for more information. The design of the network and protocol allow for multiple dashboard devices to be running and connected at once, allowing for multiple users to interact with the network at any given moment.

Sent Packets

- `ack`: Sent in response to all received packets to acknowledge successful reception.
- `new`: Sent in **Non-Broker Device Form** during initialisation to request a unique device ID
- `syn`: Sent in **Dashboard Form** once the user runs the `sync` command. This requests the broker to send all known device information back to the requesting dashboard.
- `sub`: Sent to the broker when the user runs the `sub` command, this tells the server that the dashboard wishes to subscribe to a given device group or device via its ID.
- `pub`: Sent to the broker when the user runs the `pub` command, this tells the server that the dashboard wishes to publish a value to a given device group or device via its ID.

Received Packets

- `ack`: When received, the original packet the acknowledgement belongs to is removed from the devices queue.
- `new`: Received in the **Broker Form** from the broker during device initialisation containing the dashboard's unique ID.
- `upd`: The broker forwards all `upd` packets received to any dashboard subscribed to the relevant sensor or actuator, the `<value>` parameter is stored in the relevant device's information within the dashboard.
- `syn`: Received from the broker in response to a syn command being published, these contains the device information for any given device known to the broker. One of these is received for each sensor and actuator on the network.

5.2.3 Sensor

Sensor devices are simple devices that generate some value, which for the purposes of demonstration is randomly generated, and regularly send this value to the broker. These devices could fulfill any number of practical roles in a hardware based (rather than virtualisation based) deployment provided the appropriate hardware, such as thermometers, humidity sensors, motion detectors, etc.

Sent Packets

- `ack`: Sent in response to all received packets to acknowledge successful reception.
- `new`: Sent in **Non-Broker Device Form** during initialisation to request a unique device ID
- `upd`: Sent at a regular interval containing a sensor's current data. For demonstration purposes this data is randomly generated using *Python's* `random` library.

Received Packets

- `ack`: When received, the original packet the acknowledgement belongs to is removed from the devices queue.
- `new`: Received in the **Broker Form** from the broker during device initialisation containing the sensor's unique ID.

5.2.4 Actuator

Actuator devices are similar to sensor devices, but instead of generating their values they are set by users using the `pub` command on a dashboard device. In a hardware based deployment these devices could be configured to control a number of devices, such as fans, servos, opening/closing blinds, etc.

Sent Packets

- `ack`: Sent in response to all received packets to acknowledge successful reception.
- `new`: Sent in **Non-Broker Device Form** during initialisation to request a unique device ID
- `upd`: Sent at a regular interval containing a actuator's current data.

Received Packets

- `ack`: When received, the original packet the acknowledgement belongs to is removed from the devices queue.
- `new`: Received in the **Broker Form** from the broker during device initialisation containing the actuator's unique ID.
- `pub`: Received from the broker, upon reception the actuators value is set to the `<value>` parameter from the packet.

6 Reflections

Overall I am fairly satisfied with the protocol and implementation that I have created. I feel that I fulfilled my initial design principals, however I believe there are some improvements I could make to help improve readability/user-friendliness. If I were to extend this protocol I would like to add the ability for a user to assign a name for each of the devices on the network, such as "*Front office thermometer*" or some other way to allow the user to more easily identify individual devices from each other.

Additionally, I was originally planning to implement the dashboards to host a web interface for users to interact with using the `flask` library for *Python*, however due to time restraints I was unable to get around to this and instead had to settle on using a command line interface (CLI). I am relatively happy with my command line interface and I feel that it should be relatively natural to use provided the user is somewhat familiar with unix/linux command lines. One thing that is lacking from my CLI implementation is robust enough error checking, as it I am sure that it is relatively easy for a user to crash a dashboard device by inputting a malformed argument for the `sub`/`pub` commands.

Whilst my protocol was designed from the start to handle multiple dashboard, sensor and actuator devices, it is not built to handle multiple brokers, given the time I would have liked to design it so multiple brokers could be instantiated and to have them communicate with one another to sync data, and allowing each device on the network to connect to the broker it has the lowest ping to. However I feel that this is wildly outside of the assignment's scope so I have not pursued it.

Another feature I would have liked to implement would be to allow users on the dashboard devices to send messages to one another, however in terms of functionality I did not deem this to be necessary. I believe that the current implementation of my protocol would be able to handle simple messaging easily, but it would probably be generally beneficial to add a time-stamp section to every packet sent, which would also allow the broker and dashboards to only store the most recent version of data from each device.

The last feature I would have liked to do would have been to implement some form of unsubscription in the dashboard's functionality, to allow users to opt out of receiving data from a given sensor/group after subscribing to them as currently the only way to reset the device subscriptions is by restarting the devices as these values are only stored in memory. It might perhaps be a good idea to implement some form of small persistent storage so dashboard subscriptions remain between boots.

The last remaining issue I could foresee with my implementation is that I have not tested it on any machine other than my own, however given the flexibility of *Docker* I suspect that there would be minimal compatibility issues. That said I would have really liked to test this protocol on actual devices, but I don't feel like hoarding all the devices in the household to try that out.

With all that said, I am very pleased with my final implementation and I felt like this was a super good intro into using `Docker` alongside sending communications between devices!