Setting everything up and initializing threads

First of all, when main is executed, we configure the logging options and call a function that will set up all the required elements for every thread (sockets, semaphores, etc.) and for the interaction between them.

Network thread

Before running the thread, it creates the socket that will receive the broadcast UDP packets. When the thread starts, it follows the next steps:

1. Obtain the next free node from the network queue (blocks if mutex is locked).
2. Wait until reception of a packet from the network and store it in the node.
3. Wake up elaboration thread so it processes the packet.

This is the structure of the nodes of the network queue:

```c
typedef struct nqnode_s nqnode_t;
struct nqnode_s
{
    struct sockaddr_in source;
    afdx_packet_t packet;
    time_t last_update;
};
```

Elaboration thread

Directly runs the thread, which performs the following operations:

1. Wait to be woken up by network thread (when there is a node).
2. Obtain the next node to be taken out from the network queue.
3. Update internal data corresponding to the packet processed.

Application manager thread

Before running the thread, it creates the UNIX socket to which client applications will connect and starts listening. When the thread starts running, it includes the UNIX socket in the set of managed file descriptor and follows this procedure:

1. Make use of the select system call in order to accept new connections and keep track of the current number of clients connected, to multiplex among them.
2. For every connected client (file descriptor), check if it has something to send.
   2.1. If so, receive the packet and process it.
      2.1.1. If it is a single request, immediately send a response (calls *serve_query*).
      2.1.2. If it is a register request, register the query.
      2.1.3. If it is an unregister request, unregister the query.
3. If there is any error with any application, unregister it.

Take into account that there exist a list of registered applications and, for every application, a list of registered queries. The data structures are as follow:

```c
typedef struct reg_app_s reg_app_t;
typedef struct reg_query_node_s reg_query_node_t;

struct reg_query_node_s {
    app_query_t query;
    reg_app_t *app;
    void *handler;
    reg_query_node_t *next;
};

struct reg_app_s
{
    int fd;
    reg_query_node_t *queries;
    reg_app_t *next;
};
```

Scheduler thread

The scheduler makes use of a queue with nodes ordered by ascending expiration time. New nodes (new requests of data updates from external applications) are inserted by the application manager when registering a query and also removed by the same thread when unregistering them. Nevertheless, the scheduler is in charge of keeping the queue updated and ordered, and trigger the action associated to every node, which is basically a call to the function *serve_scheduler_query*, that internally calls *serve_query*.

This is the structure of the nodes of the scheduler queue:

```c
typedef struct scheduler_node_s scheduler_node_t;
struct scheduler_node_s {
    timespec_t interval;       // How often to re-schedule
    timespec_t next_time;      // Next time to run
    void (*action)(void *);    // What to call
    void *data;                // Argument passed to action
    scheduler_node_t *prev;
    scheduler_node_t *next;
};
```

The thread keeps executing the following:

1. Check if there is any node on the queue and the current time is higher than the next execution time of the first node.
   1.1. If there is no node (empty queue), wait (block) until a node is inserted.
   1.2. If there is some node but it is not the time to execute the query yet, wait (block) until that time or until first node changes (will be woken up when this happens).
   1.3. If, instead, both conditions are satisfied, perform these operations:
      1.3.1. Get first node from the queue.
      1.3.2. Reschedule update: insert the same node with ‘next time’ field updated, in its correct position.
      1.3.3. Run action associated to that node, meaning send the update to the corresponding application.

Receiving AFDX packets from the network

This is very simple: As previously explained, the network thread is in charge of receiving the packets and store them in the network queue. Packets are broadcast, so in order to receive them we have to set the corresponding option after creating the socket; also, passing the address 0.0.0.0 allows us to listen from all IP addresses.

When a packet is received, the reception time is stamped on the network queue node and the elaboration thread is woken up.

Updating internal data

Elaboration thread pulls out packet identifier from the network queue node, which can have the following three values:

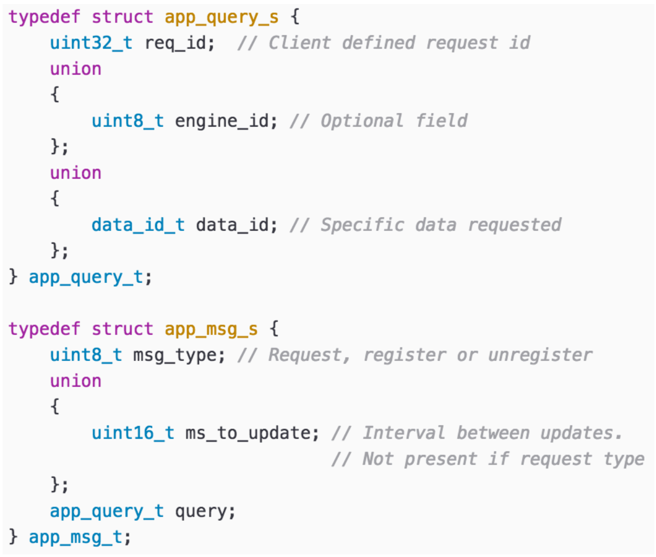
1. AFDX_PACKET_TYPE_UNKNOWN: Discard the node.
2. AFDX_PACKET_TYPE_ADIRU: Update all internal data related to attitude and speed.
3. AFDX_PACKET_TYPE_ENGINE: Retrieve engine id from the node and update all engine related internal data corresponding to that engine.

Making use of a mutex, the elaboration and network thread are synchronized, so that the elaboration thread only tries to retrieve nodes from the queue when it is not empty.

Answering single requests

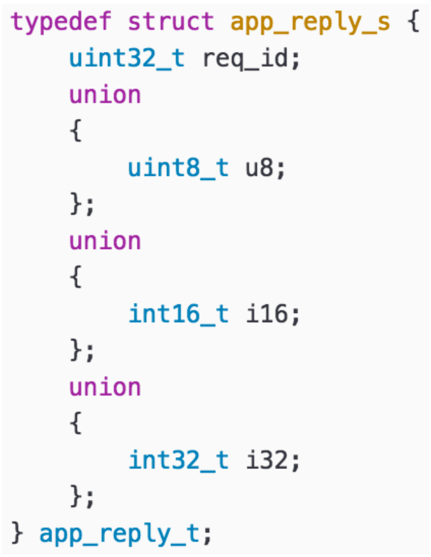
As it has been commented, when an app asks one shot for a piece of data, the application manager itself answers the request, taking a look at the data id and filling the corresponding field of the reply message.

Here we show the data structures used for both messages (request and reply):



```
typedef struct app_query_s {
    uint32_t req_id;  // Client defined request id
    union
    {
        uint8_t engine_id; // Optional field
    };
    union
    {
        data_id_t data_id; // Specific data requested
    };
} app_query_t;

typedef struct app_msg_s {
    uint8_t msg_type; // Request, register or unregister
    union
    {
        uint16_t ms_to_update; // Interval between updates.
                               // Not present if request type
    };
    app_query_t query;
} app_msg_t;
```



```
typedef struct app_reply_s {
    uint32_t req_id;
    union
    {
        uint8_t u8;
    };
    union
    {
        int16_t i16;
    };
    union
    {
        int32_t i32;
    };
} app_reply_t;
```

As can be seen, the request message works for punctual requests, register or unregister requests, thanks to the message type field and to the presence of several optional fields (milliseconds, for instance).

Registering and unregistering queries

Register query

This (and perhaps also the unregister operation) could be considered as the most complex part of the program, but the very basic steps could be summarized like this:

1. A register packet arrives, call *register_query*.
2. Update list of queries of the app (add new query). If it is the first query, add app to list of registered apps.
3. Schedule query (insert a node into scheduler queue).
4. Scheduler thread will take care of when to send message to client (it has been already explained how this thread does it).

The action field from the scheduler node is always assigned the function *serve_scheduler_query*, and the data passed is the query node that contains the data id; this way, the query can be served like when the application manager directly answers one-time requests.

Unregister query

The procedure is similar to the previous one:

1. An unregister packet arrives, call *unregister_query*.
2. Update list of queries of the app (remove query).
3. Unschedule query (remove node from scheduler queue); when we perform this action, the server will never send that update to the app again (unless it registers again).

Working VS Not working

Working

All the requested features are working correctly: receiving broadcast data from the network, storing it, responding to single requests, maintaining a set of sockets with all connected applications that want to be updated on any data and sending periodic information to those apps.

Some tests have been carried out to prove this: the server has been started, together with the UDP packets generator and four different apps, everything at the same time. Two of the apps register two queries, receive enough packets and then unregister; the other two just ask for a piece of data. Everything works as intended.

Not working

Some heavy tests have been performed, that is, setting timing of delivery of UDP packages and update intervals of the apps to very low values. In the case of update intervals, setting it to something between 1 and 5 milliseconds causes errors (sometimes) when sending the replies like 'bad file descriptor' or 'broken pipe', above all when stopping any client via Ctrl-C or running both registered apps at the same time in the worse conditions (updates every 1 millisecond); in these conditions, it also happens occasionally that the server randomly exits.

These results make us see how complicated things get when processes run very fast and synchronization is not so easy to achieve.