Universidade do Porto
FEUP Faculdade de Engenharia

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING
TELECOMMUNICATIONS, ELECTRONICS AND COMPUTERS
NETWORKS AND COMMUNICATION SERVICES

# USER-AWARE FLYING AP

**FAP Management Protocol**

**Curricular Unit of Mobile Communications**
**Professor Manuel Alberto Pereira Ricardo**

André Oliveira, up201405639
Miguel Barros, up201404278
4MIEEC_T_RC

May, 2018

# Contents

**1 Introduction** **1**

**2 Client** **1**
2.1 External Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
2.1.1 JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
2.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
2.2.1 JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
2.2.2 Client logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

**3 Server** **2**
3.1 External Libraries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
3.1.1 JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
3.1.2 MAVLink . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
3.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3.2.1 JSON . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3.2.2 Server logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**4 Conclusion** **4**

**Appendices** **5**

**A Listings** **5**

# 1 Introduction

The idea behind this project is to provide a Flying Access Point to various ground Android users. It was proposed by Eduardo Almeida @ INESC TEC.

This part of the project was divided in two major parts:

- Client
- Server

This short report follows the same approach, keeping a coherent, simple explanation for each. External libraries mentioned, followed by the notable aspects of Implementation for the Management Protocol, both Client and Server.

Implementation sections focus on JSON, since it is key in the communication method, and protocol logic itself.

# 2 Client

## 2.1 External Libraries

### 2.1.1 JSON

The **Jackson** library was used to encode JSON messages and parse JSON responses. Version 2.9.5 was used in this project.

Its homepage is available at https://github.com/FasterXML/jackson.

## 2.2 Implementation

### 2.2.1 JSON

JSON handling in this project revolves around using the *ObjectMapper* class, focusing on its ability to map a *POJO*[1] to JSON as a *String*, and vice-versa.
JSON message parameters can be translated as <**key**, **value**> pairs, which is why messages were built and decoded as Hash Maps, using the *LinkedHashMap* class since it preserves insertion order.

In the case of a nested JSON message (e.g., coordinates update), the **value** in the <**key**, **value**> pair is simply another *LinkedHashMap*.

### 2.2.2 Client logic

This section will focus on explaining the key aspects of the client side implementation logic which were not restricted/guidelined in the provided documentation.

- **Initialization** Reusable parameters/objects are initialised: the user id, the Socket and the ObjectMapper.

[1]**P**lain **O**ld **J**ava **O**bject

For the ObjectMapper, the *AUTO_ CLOSE_ SOURCE* setting must be manually turned off (false), since this would cause any JSON parsing close the Socket input stream. As a consequence, this would close the connection to the server.

- **Getting the user id** Since this is the last octet of the user's IP address, it is necessary to obtain the IP address first. This is done resorting to a static method, *getLocalHost()*, from the Java *InetAdress* class.
- **Timeouts** There are three major event types in which timeouts are set:
  - Connecting to the socket;
  - Writing to the socket;
  - Reading from the socket.

  In each case, the specified timeout value is applied.
- **Connection** Connection is established through the initialised Socket object in the association request, until terminated any time something goes wrong (i.e., an error occurs) or the Client dissociates from the FAP. Otherwise, the Socket remains connected so as to allow sending coordinate updates.

  If user association is rejected, an the error value is returned, as expected. Therefore, the higher-level application is responsible for sending a new association request.

## 3 Server

### 3.1 External Libraries

#### 3.1.1 JSON

The **Parson** library was used to encode the JSON messages and parse JSON responses in the server side. This library implements the encoding and decoding of JSON messages in a simpler way compared to the original library.

Its homepage is available at https://github.com/kgabis/parson.

#### 3.1.2 MAVLink

For this project, MAVLink v2 was provided as a means of communication between the FAP Management Protocol (Server) and the Drone Autopilot.

- **Global settings**
  1. Convenience functions should be enabled.
     (i.e. *MAVLINK_ USE_ CONVENIENCE_ FUNCTIONS* is defined)
  2. A *mavlink_ system_ t* structure should be defined in order to keep track of the System ID[2] and the Component ID[3]. These are used by the MAVLink *message_ msg_ xx_ send()* convenience functions.

[2]ID of the SENDING system. Allows to differentiate different MAVs on the same network.

[3]ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot. This allows to use MAVLink both for onboard-communication and for off-board telemetry

3. If using convenience functions, define comm_send_ch()[4] if you wish to send 1 byte at a time, or MAVLINK_SEND_UART_BYTES() to send a whole packet at a time.

- **Channels** If using a single stream, only channel 0 should be used. This is done specifying the *MAVLINK_COMM_0* constant.
- **Initialisation** The appropriate Serial port must be opened and used throughout communications.
- **Receiving** Receiving can be done resorting to the helper function *mavlink_parse_char()*, typically enclosed in a *while()* loop.
- **Transmitting** Transmitting can be done using the helper functions. The helper function for a heartbeat message would be *message_msg_heartbeat_send()*
- **Termination** The Serial port must be closed.

Sources:

https://github.com/mavlink/c_library_v2

https://mavlink.io/en/getting_started/use_source.html

https://github.com/mavlink/c_uart_interface_example

## 3.2 Implementation

### 3.2.1 JSON

The JSON library, **Parson**, uses two types of structures: *JSON_VALUE* and *JSON_OBJECT*. The first structure holds the JSON message and the second one accesses it as a JSON object. To parse a JSON it is just necessary to insert, taking in consideration the order, the parameters as <**key**, **value**> pairs.
When the JSON message is complete it is possible to convert the *JSON_OBJECT* to a *string*. To decode, the process is similar, first convert to a JSON object and then retrieve, using the name of the pair, the value wanted.

### 3.2.2 Server logic

- **Thread structure** In order to keep track of associations in the server, a thread structure array was created: *threads_clients*[5], with a size equivalent to the maximum number of accepted users. A first thread is created to wait for incoming connections. This thread then creates an individual, dedicated thread for each connection so long as there is still room for another user.
- **Alarm thread** The alarm thread is created when the clients establishes connection with the server. It controls the maximum elapsed time allowed between updates. This thread will only check the difference between the timestamp and the actual time after the first insertion in the *GpsNedCoordinates* structure. In case of an error, and if the client never sends his position and timestamp, the client's socket will be closed after

[4]See Appendix A, Listing 1.
[5]See Appendix A, Listing 2.

$1.5 \times$ *GPS_ COORDINATES_ UPDATE_ TIMEOUT_ SECONDS*. This gives enough time for the client to update his coordinates, without closing the socket, and this way it does not block the thread indefinitely.

- **Thread safety** In order to prevent concurrency problems, a *mutex* is locked whenever a function tries to evaluate and/or increase/decrease the active users counter and/or the individual thread's status variable. This last variable represents whether a certain user dedicated thread exists or not.
- **Getting all users coordinates** Regarding the implementation of the function *getAllUsersGpsNedCoordinates()*, a global *GpsNedCoordinates* structure was created, with maximum size *MAX_ ASSOCIATED_ USERS*, to simplify this process. Every time a client sends a *GPS_ COORDINATES_ UPDATE* the server converts the received RAW coordinates to NED format and inserts them in the global structure. As a result, in the *getAllUsersGpsNedCoordinates* API function we just need to verify all the positions that are not empty and copy them to the recipient array given as a parameter.
- **Timing the Heartbeat** Upon initialisation of the protocol, a dedicated thread is started to control the timing and transmission of the Heartbeat messages.
  The *timespec* structure is used (it offers microsecond precision) to request time the current time through the *clock_ gettime()* function. The type of clock is monotonic (*CLOCK_ MONOTONIC*), so it doesn't get affected by possible time skipping in the OS.

  The message is then sent and another time sample is taken. The elapsed time is calculated and compared to $0.5s$. If the remaining time is equal to or smaller than zero, the thread restarts this process. If not, it sleeps for the remainder of the time, using *nanosleep()*.
- **Termination** The *terminateFapManagementProtocol()* function activates an exit flag that is responsible for closing all clients' connections. It then waits for all threads to terminate. This function is also responsible for stopping the Heartbeat handler, through a similar process which activates an exit flag dedicated to the Heartbeat.

## 4 Conclusion

This project got the expected results for both Client and Server. The Client's test sometimes returns errors (either two or three) due to the fact that the randomly generated coordinates may have a distance higher than the maximum allowed (300 m). In this scenario, the server is expected to gracefully terminate the connection, hence causing consequent client requests to fail.

The provided Client's test source code isn't a good representation of the reality of a multi-user environment, since the program generates an ID for a user based on the IP address's last octet (which is expected to be unique). Some client distinction is, however, assured in the messages the Server prints to the standard output.

# Appendices

## A Listings

```
#include "mavlink_types.h"

void comm_send_ch(mavlink_channel_t chan, uint8_t ch) {
    if (chan == MAVLINK_COMM_0) {
        uart0_transmit(ch);
    }
    if (chan == MAVLINK_COMM_1) {
        uart1_transmit(ch);
    }
}
```

Listing 1: Example implementation of the *comm_send_ch()* MAVLink helper function

```
typedef struct threads_clients {

    pthread_t  tid;         // pthread_t for each client
    int        socket;      // Socket information
    int        status;      // Informs if the thread is active or not
    int        user_id;     // User id
    int        alarm_flag;  // Flag responsible to close the client

} threads_clients;
```

Listing 2: Structure created for keeping information regarding clients