iPad/iPhone App as a Front-end for Prototype of a Highly Adaptive and Mobile Communication Network using Unmanned Aerial Vehicules (UAVs)

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Adrian Hänni
2014

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Abstract

Wireless Mesh Networks (WMNs) have made a rapid progress over the past few years and have inspired numerous developers since they provide an highly flexible communication infrastructure. The concepts of WMNs provide new scopes and use cases in daily life as well as after a severe disaster. In disaster scenarios, it is urgent to have a working communication infrastructure. The architecture of a WMN allows to deploy a required network in a fast and reliable way. Therefore, the prototype *UAVNet* was developed. It is a framework that provides an highly adaptive, autonoumous, airborne deployment of a WMN using Unmanned Aerial Vehicles (UAVs) and is intended to be used in future disaster scenarios. *UAVNet* provides different deployment procedures using a single or multiple UAVs.

Such a system must be manageable in an usable way to take the steps towards a real-life operation. Hence, the *Remote Control App*, a front-end application for the *UAVNet* was developed. It monitors the current states of the UAVs and is used to set up and to abort a deployment scenario for the *UAVNet*. It was developed as an user-friendly mobile application for iOS devices and additional features are supported to improve user experience. This includes saving of the provided information in the *UAVNet*, the use of different map sources and exportable data. The application can monitor a deployment scenario with a single UAV or multiple participating UAVs and provides the functionality to operate in an environment without Internet access. The UAVs of the deployed network are represented and monitored on an interactive map using either locally stored or online available map data sources. Further, the application's setup and managing procedures for the deployment of the WMN is easy to perform. This assures that end-users can understand its functionality in a short time since the time spent is important in disaster scenarios. The deployment process using the *Remote Control App* was successfully tested for the supported deployment scenarios in an outdoor environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few years mobile technology has evolved dramatically in consequence of the increased popularity of mobile devices. Nowadays, smartphones and tablet devices are very prevalent and the use of mobile applications has increased accordingly. The expanded availability of various mobile applications for different domains and purposes assist us in daily life situations. Mobile applications fullfill our demand for information, communication and entertainment. They cover many versatile use cases, are benefical for everday activities and take advantage of the possibilities provided by increased mobile Internet access as well.

Mobile devices provide a platform to manage highly mobile, deployable systems such as the ”UAVNet: A Prototype of a Highly Adaptive and Mobile Wireless Mesh Network Using Unmanned Aerial Vehicles (UAVs)”[1], which was implemented by Simon Morgenthaler as his Master’s thesis. *UAVNet* is a concept of an airborne, autonomously deployable, transient Wireless Mesh Network (WMN) carried by UAVs. The UAVs establish an airborne WMN for a certain size of an area as they are carrying the WMN nodes. In this thesis, the front-end application termed *Remote Control App* for the *UAVNet* is implemented, introduced, described and tested.

This chapter discusses the motivation behind *UAVNet* and its front-end application in Section 1.1, followed by an introduction on the iOS operating system in Section 1.2. Section 1.3 gives an overview about WMNs. Afterwards, a definition and a short description for UAVs is presented in Section 1.4. The subsequent chapters are described in Section 1.5.

1.1 Motivation

Because of increasing human settlements, population growth and global effects as climate change, severe disasters affect more people. In disaster scenarios, such as floods, earthquakes, avalanches and storms, the communication structure of the affected area may become inoperative and damaged, which results in a discontinuous communication chain. Most communication infrastructure is ground-based and its hubs may become inaccessible due to the effects of a disaster on the ground. For the rescue forces, it is essential to have an operating communication infrastructure for coordinating the help and gathering the needed informations. It is a challenge to establish a ground-based communication infrastructure in a destroyed or badly accessible area. Therefore, an air-based solution provides additional benefits, such as better accessibility, avoidance of mechanical challenges and better coverage. In respect of these facts, Simon Morgenthaler implemented a prototype for deploying an autonomous, highly adaptive, airborne Wireless Mesh Network (WMN) using UAVs[1].

We assume that local people, who are responsible for the deployment of the airborne WMN in the disaster area, may not be technically skilled. Accordingly, it is required that the front-end application has a good usability and is mostly self-explanatory. The front-end application should offer the required functionality to quickly set up the deployment of a WMN. The usage of mobile devices such as smartphones and tablets as platform for the application is reasonable because of the fact that they are widely spreaded and their owners are familiar with them. For this project the iOS operating system was chosen which runs on Apple's iPhone, iPad and iPod devices. The application makes use of the location-awareness of these devices as well as of the UAVs and provides an integration of different maps. Hence, the availability of an application providing both usability and functionality is a key for establishing an airborne WMN by rescue forces.

1.2 iOS

The proprietary operation system iPhone OS was first presented in January 2007[2]. It is an adapted operating system for mobile devices based on Mac OS X which includes support for touch-gestures. Since its operation has been expanded to Apple's iPod and iPad devices, the iPhone OS was renamed to iOS in June 2010[3]. iOS operates only on iPhone, iPad and iPod devices due to security as well as commercial considerations and except for a few open-source components, it is a closed system. Native application for iOS are implemented in Objective-C and are only available over Apple's App Store. The iOS applications are implemented with the free available application *Xcode* that provides a full integrated development enviroment (IDE) including a device simulator for developers.

There are four abstraction layers given in iOS termed Core OS, Core Services, Media and Cocoa Touch[4]. Each of these layers is made up of different frameworks that can be used and incorporated into the applications. The stack of these layers is illustrated in Figure 1.1.

- The Core OS layer is the foundation of the operating system. It is responsible for interaction with the hardware as well as taking care of file system tasks, security, memory management, threats and networking.
- The Core Services layer provides an abstraction of the services offered by the underlying Core OS layer including technologies and frameworks to support features as Core Location, iCloud, Core Data, Address Book, SQLite, XML support, and others.
- The Media layer offers multimedia services including graphic, audio and video technologies as well as the proprietary streaming protocol "AirPlay".
- The Cocoa Touch layer contains key frameworks for the user-interface and focuses on optimization for touch-based interfaces. It consists of many high-level system services

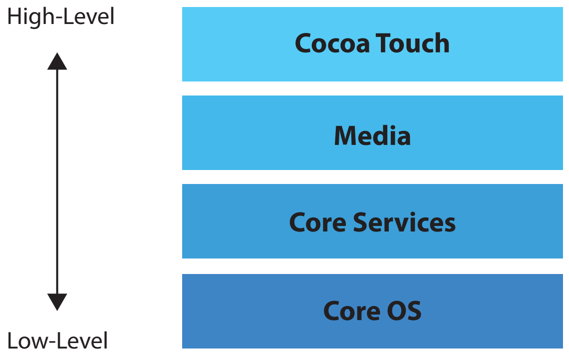


Figure 1.1: Stack of iOS abstraction layers

as touch-based input, push notifications, multitasking, animation, components for views and graphical interfaces. Its design is constructed according to a Model-View-Controller (MVC) software architecture and is based on the Mac OS X Cocoa Application Programming Interface (API).

Modern iOS mobile devices come up with touchscreen, location service, built-in accelerometer and other functionalities. Their hardware specifications provide possibilities for extended purposes of use, but there are some limitations of iOS platforms a developer has to be aware of[4].

Limitations of the Device Simulator
The device simulator is acting close to the real device concerning hardware and software compability, but it does not behave identical. The applications running on the simulator could run without failures while the same application fails on the mobile device as the same frameworks available on the simulator and the device have a few differences. Even for the really emulated APIs some differences in behavior can occur, since the simulator uses several OS X frameworks as part of its own implementation and Cocoa frameworks from Mac OS X. The execution of processes is more performant in the simulator compared to several different types of iOS devices. Additionaly, the simulator misses some hardware functionalities such as the camera and the digital compass.

Limitations of Storage Capacity
iOS only requires a few hundred MB for storage and it offers frameworks covering a large-scale of purposes. Those frameworks, built-up from precompiled routines, faciliate the execution and development of applications. The system files are tailored to fit into a small storage. Although, the free storage capacity depends on the saved content and the device model, the iOS applications can be as large as 2 GB, but the executable file cannot exceed 60 MB.

Limitations of Main Memory
Memory management for iOS applications does not use garbage collection and is based

on a reference counting model. The applications cannot use a page file for extending the physical memory. By using a reference counting model for the memory, the developer has to assure that the application frees up data that is no longer in use. Keeping unneeded data can lead to memory leaks that consume ever-increasing amounts of memory. If the amount of free memory reaches a certain threshold, the operating system requests the applications to release memory voluntarily. Applications that fail to release enough memory are terminated. Overall, the memory limit varies depending on the currently running applications and the device model.

Limit of Data Access

Every iOS application runs in a sandbox. This means the iOS applications do not have access to certain resources and run in a regulated environment of the operating system. iOS installs each application in its own sandbox directory which acts as the home for the application and its data. Each sandbox directory contains several well-known subdirectories for placing files. As a result of the sandbox processing, an application does not have access to files of an other application. The purpose of a sandbox is to limit potentional unwanted effects that a compromised application can cause to other applications and the system.

Limit of Interaction

The absence of physical input devices does not mean that users have to loose flexibility in interaction with the device, but adaptations have to be made. Instead of imitating a desktop enviroment, the design should provide a easy-to-use interface that takes care of usability enhancements for mobile devices, such as touch gestures, device shaking, adaptive layouts and motion gestures.

Limit of Energy

Energy consumption cannot be ignored on mobile devices. Since high CPU usage can lead to a rapid decrease of energy reserve, the developer has to be cautious on performing long-term calculation tasks.

Limitations for Applications

Each application available on the Apple App Store has to go through an approval process for being officially released. The violation of certain guidelines[5] cause an application to be rejected. For example, applications that fundamentally duplicate the built-in iOS applications without adding new functionality. Furthermore, there are some restrictions on functionalities that an application can provide. An application cannot run in the background, nor can it run besides of an other running application. Although, Apple introduced mulitasking for iOS, it cannot be understood as multitasking in the usual way. Only certain tasks or services can run in the background either for a finite period or periodically and the developers have to implement explicit support for the use of multitasking features.

Limitations by User Actions

Most iOS applications are used sporadically. An application design should take care of the nature of a common user who will use an application only over a short interaction period. The presence of complex interfaces and too many settings confuse and should be avoided.

1.3 Wireless Mesh Networks

A WMN defined by the standard IEEE 802.11s is considered to be a special case of Mobile Ad-hoc Networks (MANETs), which stands out by interconnection of radio nodes organized in a mesh topology. Each node in a WMN acts as a host as well as a router for multihop destinations. In WMNs, there can be both fixed and mobile nodes, where each node has the same functionality and responsibility according to the network topology. A WMN is dynamically self-organized, self-healing and self-configured. The meshed topology offers a way to avoid the centralized approach of traditional wireless network architectures and it ensures the possibility of multiple routing paths for each node in the network. If one or multiple nodes fail, the packets will be rerouted taking a different path in the communication chain. The reliability of the communication and the network coverage are increased by deploying additional nodes. Mobile clients, which are connected to a WMN, can roam within the network coverage area and have untethered connectivity across different mesh nodes. Due to the meshed structure, a WMN supplies good reliability, network coverage, scailability as well as low upfront investments[6]. Furthermore, WMNs can interconnect to other networks such as IEEE 802.11, IEEE 802.15, IEEE 802.16, Internet, cellular, Worldwide Interoperability for Microwave Access (WiMAX) and Wireless Sensor Network (WSNs) through the gateway and bridging functions in the mesh routers[7]. A scheme of a WMN is shown in Figure 1.2.

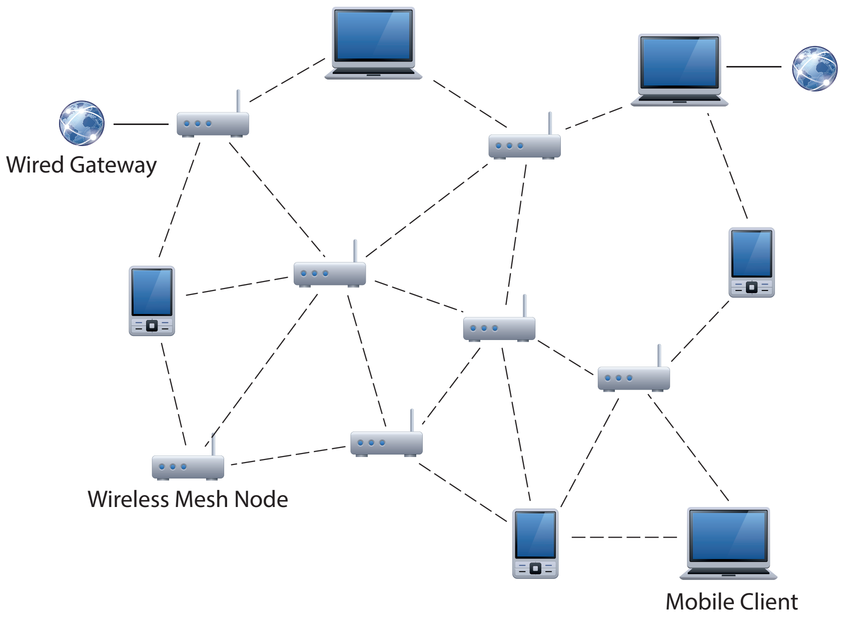


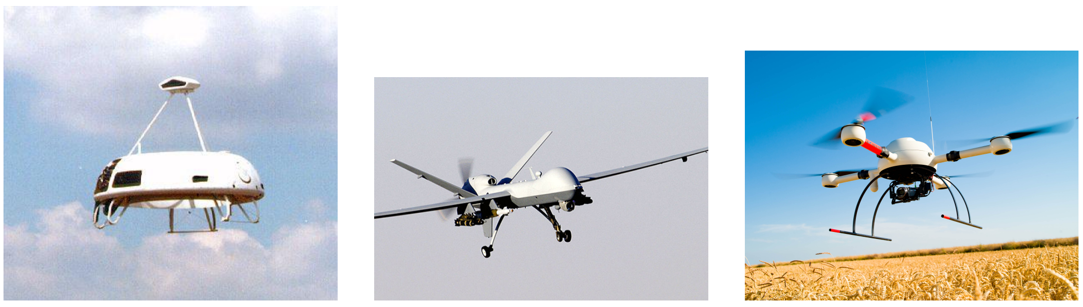
Figure 1.2: Wireless Mesh Network

According to [7, 8, 9], WMNs are considered to be promising candidates for the next-generation wireless broadband networking. Because of their advantages in respect to traditional wireless formats such as the absence of a rigid structure and the flexible deployment model, it is expected that WMNs will be part of upcoming wireless network applications as they are suitable for many different kinds of scenarios.

Although, WMNs provide inherent failure tolerance through multi-path routing, rapid deployment and flexible coverage areas[9], there are some concerns regarding to scalability both in terms of number of clients and increased area of network coverage[8]. The increase of network capacity is incompatible with the improvement of the coverage due to the factors such as multi-hop routing, which degrades the performance with increasing number of nodes. The usage of IEEE 802.11 MAC protocol, which is not scalable, degrades throughput as the number of hops increases[9]. Other critical performance factors are the state of the art of radio techniques, the behavior of mesh connectivity, broadband and QoS, compatibility and inter-operability of conventional and mesh clients, security, and the ease of use[7].

1.4 Unmanned Aerial Vehicles

An Unmanned Aerial Vehicle can be defined as ”a powered, aerial vehicle that does not carry a human operator, uses aerodynamic forces to provide vehicle lift, can fly autonomously or be piloted remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload”[10]. Although UAVs are mostly known for their military usage nowadays, they can also perform surveillance, communication relay, remote sensing, search and rescue, scientific, commercial and public safety tasks. Modern UAVs have a built-in flight control and guidance assistance which allows them to carry out simple flight commands as waypoint following, flight-path stabilization and calculating, velocity and altitude control. To perform those flight procedures, UAVs have recourse to sensor data from multiple sources like gyroscope, GPS, lasers, airpressure sensors and others. There are a wide variety of UAV shapes. Some examples are shown in Figure 1.3.


(a) Cypher UAV, Author: Wikipedia, Public Domain

(b) MQ-9 Reaper, Author: Pacific Standard, Public Domain

(c) MD4-1000, Author: micro-drones.com, Press Image

Figure 1.3: Examples of UAVs

1.5 Structure of the Thesis

Chapter 2 provides an overview about the concepts of the *UAVNet* where its basic architecture, the different types of messages and deployment scenarios as well as the simulator used by the *Remote Control App* are introduced. Then, the functionality of the *Route-Me* library is explained including the map database creation and its caching system. The front-end application *Remote Control App* of the *UAVNet* is presented in Chapter 3. It expresses the architecture, implementation, usage and design of the *Remote Control App* including the interaction with the *UAVNet*. Chapter 4 concludes this thesis. It points out to possible future extensions and use cases.

Chapter 2

Related Work

For the interaction with the *UAVNet*[11], its concept and operation has to be understood. It provides different deployment scenarios and there are several well-defined types of network messages which an application can use to control and monitor the *UAVNet*. For the integration of maps, the *Remote Control App* uses the free available *Route-Me* library[12] due to its additional benefits. In contrast to the MapKit framework provided by the Cocoa Touch layer from the iOS operating system, it supports various map sources and the ability to display locally stored map sources as well.

This chapter gives an overview about the architecture, communication and supported network scenarios of *UAVNet* in Section 2.1. It leaves out background information that is not fundamental necessary for a front-end application, such as the details about the hardware components of the *UAVNet*. For more detailed information, the consideration of "UAVNet: A Prototype of a Highly Adaptive and Mobile Wireless Mesh Network Using Unmanned Aerial Vehicles (UAVs)"[1] is recommended. The description of the *Route-Me* library in Section 2.2 concludes this chapter.

2.1 UAVNet

UAVNet is a framework of a prototype that is capable to deploy an autonomous, highly adaptive, airborne Wireless Mesh Network (WMN) carried by UAVs. It is intended to be used in disaster scenarios, such as earthquakes, floodings and avalanches, when ground-based communication infrastructures have stopped working or were never available. A disaster could have severe effects on the infrastructures on the ground, such as impassable roads, electrical blackouts, disconnected communication hubs and other damages. As the use of UAVs enables an autonomous, airborne deployment of WMN nodes, this concept grants additional advantages compared to a ground-based autonomous deployment. The UAVs can reach certain locations that are no longer accessible in a practial way. It can be assumed that deployment over the air can be achieved faster than deployment on the ground since mechanical barriers on the ground can be avoided. Because radio waves of frequencies used by WLAN are susceptible to physical obstacles, flying mesh nodes are more favorably placed than mesh nodes on the ground. Hence, the mesh nodes of the *UAVNet* can provide a better network coverage.

First response rescue forces usually have a lack of information. As a consequence, they require to exchange reports and information with responsibles, residents and among each other. A WMN is capable to provide a network for multimedia data transmissions. The WMN provides a temporary solution for the communication within its coverage. If a mesh client can provide a gateway to a network that is connected to the Internet, even world-wide communication would be possible for the nodes of *UAVNet*. Therefore, the *UAVNet* WMN can improve the ability to communicate with rescue forces in a fast and reliable way. This benefit will raise the chance for a successful coordination of the rescue forces within the disaster area.

The general deployment process of the *UAVNet* uses two client notebooks running the *UAVNet* client software. The first client is known at the beginning of the deployment by the participating UAVs and the second has to be discovered for establishing the WMN between these clients. The supported scenarios for the deployment of the *UAVNet* are described in Subsection 2.1.4.

2.1.1 Architecture

UAVNet consists of client devices, wireless mesh nodes and UAVs. Its typical setup is shown in Figure 2.1. The clients can either be notebooks, which communicate with each other over the WMN, or mobile devices, such as iPhone, iPad and iPod, which configure and monitor the *UAVNet*. *UAVNet* provides access over the IEEE 802.11g standard for the clients because most mobile clients do not yet support the IEEE 802.11s standard for wireless mesh networks.

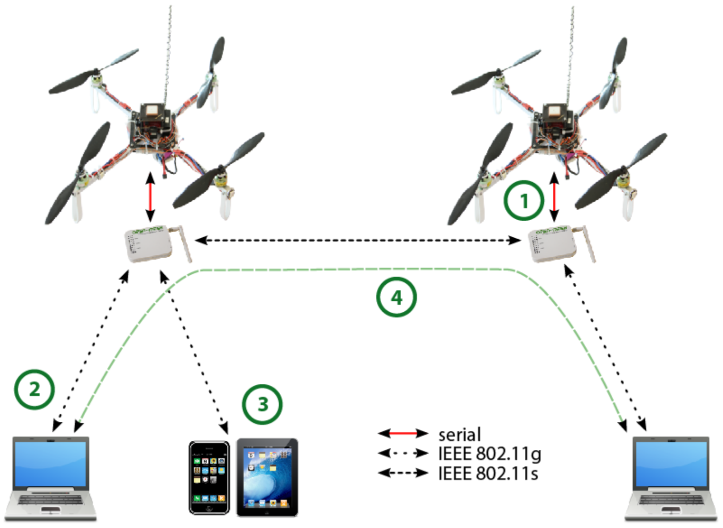


Figure 2.1: Typical setup of *UAVNet* with two UAVs with attached wireless mesh nodes and multiple clients.[1]

The different connections are described as follows[1]:

1. The wireless mesh nodes are directly connected to the flight electronics of the UAVs using a serial connection.

2. The two client notebooks of the rescuers are connected to the mesh network using a standard IEEE 802.11g wireless connection. The wireless mesh nodes act as ordinary APs.

3. Mobile devices like iPhone and iPad are used to configure the network and monitor the UAVs as well as and the client notebooks. They use also an IEEE 802.11g wireless connection to interact with *UAVNet*.

4. Traffic between end devices is forwarded over the IEEE 802.11s wireless mesh network, automatically set up by the UAVs.

2.1.2 Uavcontroller

The *uavcontroller* is the primary software component of the *UAVNet*. It is implemented with the C programming language and runs as a daemon on every wireless mesh node. It consists of a main part, several threads, as well as the *libuavint* and *libuavext* libraries. It is responsible to handle the communication between the mesh nodes, the clients and the UAV electronics. The scheme of its architecture is illustrated in Figure 2.2.

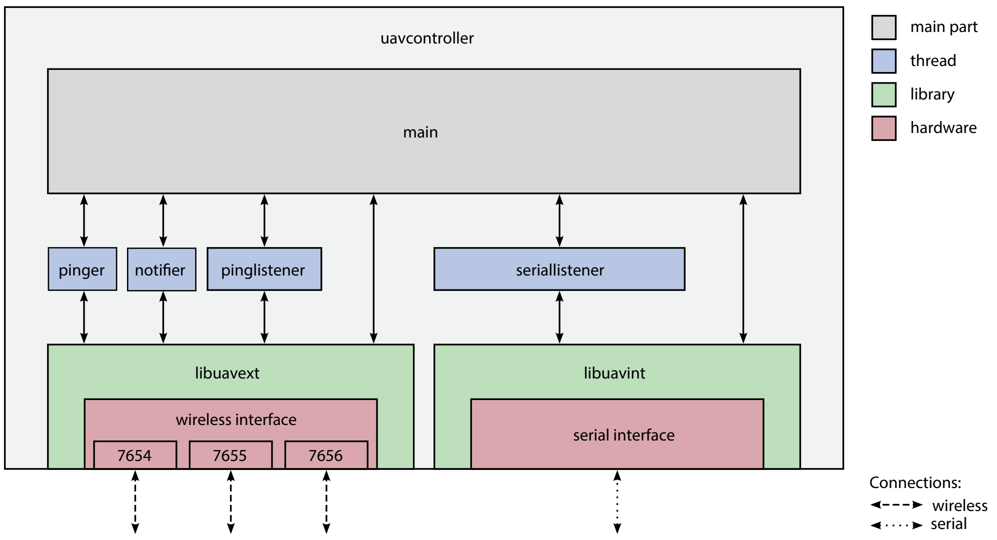


Figure 2.2: Scheme of the architecture of the *uavcontroller*[1]

The *main* part of the *uavcontroller* manages the *pinger*, *notifier*, *pinglistener* and *seriallistener* threads, the communication over the serial and wireless interface and the flight path of its controlled UAV. It contains the logic behind the different types of network scenarios and the desired behavior of the UAVs. Its methods and algorithms ensure that every UAV acts as intended, depending on the current state of other UAVs and the state of the network deployment. The *main* part handles processing and evaluation of received messages and keeps track of the status of discovered clients, managing and monitoring devices, and other UAVs. It uses the *libuavint* library for the serial communication with the UAV electronics and the *libuavext* library for the external wireless communication, providing both required methods. Both libraries have a similar construction and are capable to handle different types of messages. These are the *control*, *ping* and *notification* messages for external communication. The libraries can be described as follows:

libuavext

libuavext is a library that is integrated in *uavcontroller* and the *Remote Control App*. Its purpose is to handle the external communication between the mesh nodes as well as between the mesh nodes and the client devices. Therefore, it provides the functionality to establish, to close and to manage sockets for TCP and UDP packets on different ports. As this library is responsible for incoming and outgoing network traffic, it contains the methods to send, to receive and to handle different types of messages. The structure of these messages require a Cyclic Redundancy Check (CRC) of the modified-base64 encoded payload. The functionality for the validation and computation of CRC values and for the encoding and decoding of payloads is included as well. Additionally, it handles common network tasks, such as hostname translation, Domain Name System (DNS) resolving and the Address Resolution Protocol (ARP).

libuavint

The *libuavint* library is needed to handle internal communication between the UAV electronics and the wireless mesh node. These two components are interconnected over a serial interface. The *libuavint* library provides the functionality to manage the communication over the serial connection including the methods to send, to receive, to handle and to validate internal sent messages. The internal messages are used to receive sensor data and to send flight commands, such as waypoints, to the UAV.

uavcontroller runs three different threads for the external communication and one for the internal communication. These threads are termed *pinger*, *notifier*, *pinglistener* and *seriallistener* and are described as follows:

pinger

The *pinger* thread does periodically broadcast a *ping* message every few seconds using port 7655. Its purpose is the announcement of the current status of the transmitting UAV to other participants within the *UAVNet*. As a consequence, the detection of formerly unknown UAVs and clients can be achieved by listening to *ping* messages. The status information of a UAV includes the hostname, the position, the IPv4 and MAC address. Additionally, they contain information about the current network scenario and the position of already discovered clients.

notifier

The *notifier* thread sends periodically unicast *notification* messages using port 7656. Each UAV has a notification service provided by the *uavcontroller* and executed by the *notifier* thread. The notification messages contain detailed information about the current values of sensor data from the UAV electronics including the height, battery, heading and speed. A managing or monitoring device can subscribe to this service and will, from this point on, receive notifications. Every UAV supports multiple subscribed devices. The subscription has to be made for every UAV separately because the UAVs do not inform each other about their subscribed clients.

pinglistener

The *pinglistener* thread handles incoming *ping* messages on port 7655. It observes the status of all connected UAVs and the detected clients. Therefore, the *uavcontroller* is aware about the current positions of those participants. If a UAV receives new information about a client that was found by an other UAV, it will update the client information in its own *ping* messages.

seriallistener

The *seriallistener* thread handles incoming messages from the serial interface and forwards them to the main part of the *uavcontroller* where the information is processed. As an example, the current values of sensors provided by the UAV electronics are received over the serial interface and are processed afterwards.

2.1.3 External Communication

Although, there is internal communcation between the wireless mesh nodes running the *uavcontroller* software and the UAV electronics, only the external communication has to be investigated for the configuration and monitoring of the *UAVNet* through the *Remote Control App*. The *libuavext* library consists of well-defined methods and message structures for the interaction between the *Remote Control App*, the client notebooks and the UAVs. The mobile device, which runs the *Remote Control App*, does not have to be set up in a special way for the communication with the *UAVNet* because each wireless mesh node runs a Dynamic Host Configuration Protocol (DHCP) service to assign IP addresses. By contrast, the wireless mesh nodes of the UAVs use static assigned IP addresses.

Types of Messages

As seen in Figure 2.2, the wireless interface of the *libuavext* library uses different ports for the *control*, *ping* and *notification* messages. Depending of their relevance, either the TCP or UDP protocol is used.

Control Messages

control messages are transmitted as unicast messages over TCP socket on port 7654. They are used to transmit subscription information for the notification service as well as to set up and to abort a network scenario. The use of TCP assures that the sending device gets informed about the success of the transmission since this information is necessary for managing applications. The *control* messages used by the *Remote Control App* are the *submitStartConfiguration*, *sendAbort*, *outSocketResponse*, *sendFlightDirection* and *sendNotificationSubscription* messages.

Ping Messages

ping messages are transmitted as broadcast messages over UDP socket on port 7655. Listening applications and other active UAVs use the information contained in *ping* messages to discover UAVs and clients. They are sent periodically as broadcast messages because some participants of the network might not be known in the current state of the deployment and there might be multiple participants waiting for the same information. UDP is used for this type of message since there is a possibility that a UAV may get out of range from client notebooks, the managing device running the front-end application and other UAVs during its flight. Since they are sent periodically and contain only general information, it is uncritical for the managing and monitoring applications if some *ping* messages get lost.

Notification Messages

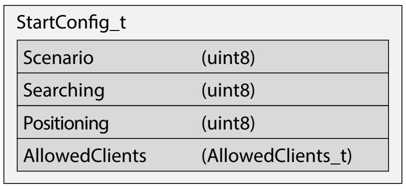
notification messages are transmitted as unicast messages over UDP socket on port 7656. They are only transmitted to the subscribed devices. It uses UDP because the UAV may get out of range from the subscribed devices, but it is not critical for the deployment of the network scenario if some messages are lost. The purpose of this kind of message is to inform about the current state of a UAV. The *Remote Control App* uses the information of *notification* messages for graphical representations of the UAVs. The consequence of lost *notification* messages is unactual information on the managing and monitoring devices.

External Messages

The deployment of *UAVNet* requires the transmission of several external messages between the *Remote Control App*, UAVs and the client notebooks. Each type of message consists of a start and end delimiter, a 1-byte command sign to classify its type, a modified-base64 encoded payload and a CRC value of two bytes[1]. The CRC procedure assures the validity of the data. As mentioned in Subsection 2.1.2, the *libuavext* library provides the methods for the encrypting, decrypting, encoding and decoding procedures. To provide transmission of complex data, such as navigation and position information, the data is packed and handled in C structures. These messages are described as follows:

submitStartConfiguration

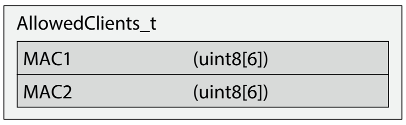
The *submitStartConfiguration* message is of type *control*. This message is used by the *Remote Control App* to submit the parameters for a desired network scenario. The deployment process starts, if this message was successfully transmitted to a UAV. It contains the *StartConfig_t* structure as payload, shown in Figure 2.3. This structure consists of the setup information of a network scenario, such as the scenario type, the searching and positioning mode for the UAVs, and the allowed clients. The *AllowedClients_t* structure is shown in Figure 2.4. It provides the MAC addresses of the two allowed clients and is used for the *Airborne Relay* and *Airborne Multihop Relay* scenarios. The integration of the allowed clients into the start configuration assures that the UAV will only accept messages of these two clients for the deployment process. If a UAV has already received a *submitStartConfiguration* message, it will not accept a second message of this type. The deployment has to be aborted using the *sendAbort* message to change the start configuration.



StartConfig_t

Scenario	(uint8)
Searching	(uint8)
Positioning	(uint8)
AllowedClients	(AllowedClients_t)

Figure 2.3: *StartConfig_t* structure[1]



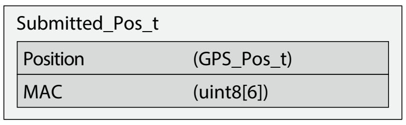
AllowedClients_t

MAC1	(uint8[6])
MAC2	(uint8[6])

Figure 2.4: *AllowedClients_t* structure[1]

submitPosition

The *submitPosition* message is a *control* message. When a UAV gets in signal range of a client, the client uses this message to submit its own position. The message contains the *Submitted_Pos_t* structure, as shown in Figure 2.5. This structure consits of the MAC address of the transmitting client and its GPS position provided by the *GPS_Pos_t* structure, as illustrated in Figure 2.6. After a UAV has received a client position, it includes this information into its regularly broadcasted *ping* messages. This mechanism ensures that the position of detected clients is propagated to other participants in the *UAVNet*.



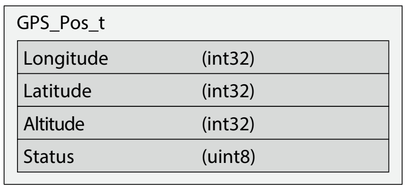
Submitted_Pos_t

Position	(GPS_Pos_t)
MAC	(uint8[6])

Figure 2.5: *Submitted_Pos_t* structure[1]

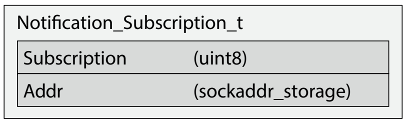
sendNotificationSubscription

The *sendNotificationSubscription* message is a *control* message. This type of message is used by the *Remote Control App* to subscribe to the notification service of a UAVs. It contains the *NotificationSubscription_t* structure, as shown in Figure 2.7, and provides the IPv4 or IPv6 address of the transmitting device. The value of the subscription integer indicates whether a device wants to subscribe or to unsubscribe.



GPS_Pos_t	
Longitude	(int32)
Latitude	(int32)
Altitude	(int32)
Status	(uint8)

Figure 2.6: *GPS_Pos_t* structure[1]



Notification_Subscription_t	
Subscription	(uint8)
Addr	(sockaddr_storage)

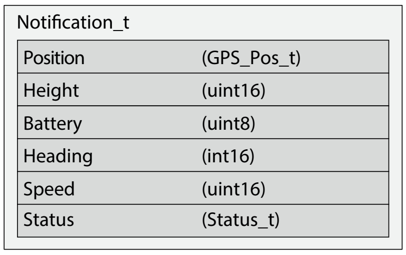
Figure 2.7: *NotificationSubscription_t* structure[1]

sendFlightDirection

The *sendFlightDirection* message is used by the *Remote Control App* to submit the GPS coordinates of a waypoint. It is a *control* message as its successful transmission requires a response to the sending application. The transmission of this message is only needed if the selected network scenario uses the *Manual Search Mode*. The GPS position of the waypoint defines the endpoint of the UAV's flight path for the discovery of the second client. The message contains the *Submitted_Pos_t* structure, as shown in Figure 2.5, that provides a format for the position.

sendNotification

The *sendNotification* message is sent regularly to all subscribed devices by the *notifier* thread of the *uavcontroller* running on the wireless mesh nodes. It defines the *notification* message type and it contains the *Notification_t* structure as shown in Figure 2.8. The *sendNotification* message provides status information included as *Status_t* structure as well as information about the current values of the position, height, battery, heading and speed. The *Status_t* structure is illustrated in Figure 2.9.

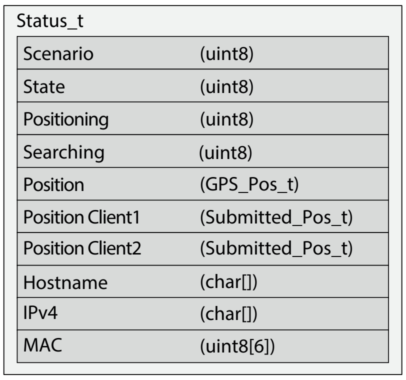


Notification_t	
Position	(GPS_Pos_t)
Height	(uint16)
Battery	(uint8)
Heading	(int16)
Speed	(uint16)
Status	(Status_t)

Figure 2.8: *Notification_t* structure[1]

sendPing

The *sendPing* message is regularly broadcasted by the *pinger* thread of the *uavcontroller*. The payload of this *ping* message is a *Status_t* structure that contains information about the type of the current network scenario, the position of the sending UAV, the already discovered clients, as well as the UAV's hostname, its IPv4 and MAC address.



Status_t

Scenario	(uint8)
State	(uint8)
Positioning	(uint8)
Searching	(uint8)
Position	(GPS_Pos_t)
Position Client1	(Submitted_Pos_t)
Position Client2	(Submitted_Pos_t)
Hostname	(char[])
IPv4	(char[])
MAC	(uint8[6])

Figure 2.9: *Status_t* structure[1]

outSocketResponse

The *outSocketResponse* message is used as response to various *control* messages. The *Remote Control App* and other transmitting clients use this kind of message to determine if a command was processed and received successfully. Therefore, the payload contains a 1-byte integer to indicate the success of the transmittion or to report that additional information is required.

sendAbort

The *sendAbort* message is a *control* message. It is used by the *Remote Control App* to signal the end of a network deployment. It simply contains a 1-byte integer as payload. After a UAV receives this message, it stops the deployment of the WMN and flies back to its start location.

2.1.4 Network Scenarios

There are three deployment procedures for *UAVNet* with different search and position behaviors. The two relay scenarios, termed *Airborne Relay* and *Airborne Multihop Relay*, provide a solution to deploy wireless mesh nodes carried by UAVs between two distant clients, which are are too far away to communicate with each other. The attached wireless mesh nodes of the UAVs provide an airborne WMN and the ability to communicate with the participants of the WMN. When the UAVs have reached their final position, a communication path between the two clients is available. Depending on the distance between the clients and the signal strength of the participants of the WMN, the use of one or multiple UAVs is required. The remaining scenario is termed *Network Area Coverage* that could provide a WMN coverage for a certain area, but it is not implemented completely by the *UAVNet*.

Both relay scenarios use the starting UAV for the detection of the second client. If a third client comes into transmission range of the searching UAV earlier than the second client, the deployment of the WMN could fail unexpectedly. Therefore, the *UAVNet* provides a mechanism to ensure that the deployment process of the relay scenarios will not get compromised. As a part of the *submitStartConfiguration* message that initializes the network deployment, the *uavcontroller* receives the MAC addresses of the allowed clients. This procedure ensures that the transmissions from unauthorised clients are ignored, but it requires that the MAC addresses are known in advance.

There are two different position procedures for the UAV, either by the geographical distance between the two clients or similar signal strength in reference to both clients. These are termed *Location Position Mode* and *Signal Strength Position Mode*. The current implementation of the *UAVNet* allows to choose these modes in the start configuration for the *Airborne Relay* scenario. The *Airborne Multihop Relay* scenario has always the same procedure for the positioning of the UAVs since it uses the functionalities of both procedures. Additionally, *UAVNet* provides two different search modes for the detection of the second client, either by a manual or autonomous procedure. Therefore, they are termed *Manual Search Mode* and *Autonomous Search Mode*. The *Airborne Relay* as well as the *Airborne Multihop Relay* scenario support these detection modes. The values for the position and search modes are parts of the *submitStartConfiguration* message. These modes are described as follows:

Location Position Mode

The *Location Position Mode* uses the GPS coordinates of both clients. When the second client gets discovered, the *uavcontroller* receives its GPS coordinates from a *submitPosition* message. Afterwards, the *uavcontroller* calculates the average of the longitude and latitude value from both clients to find the geographical center between them. Because the result is processed as the next waypoint to the UAV electronics, the UAV will position itself between the two clients.

Signal Strength Position Mode

The *Signal Strength Position Mode* extends the *Location Position Mode* for the adjustement of the UAV's position. In the first place, the UAV is always positioned using the *Location Position Mode*. Then, the *Signal Strength Position Mode* is performed. It uses permanent measurements of the signal strength from both clients and directs the UAV towards the client with the lower signal strength until the signal strength in reference to both clients is equal. The mode ignores values that are expected to be wrong. As measured values of signal strengths fluctuate, it would cause the UAV to move constantly forward and backward. To avoid permanent movement of the UAV, this mode uses a threshold value to determine the equality of the signal strengths. The UAV will finalize its position, when both signal values have the same value in reference to a certain range defined by the threshold.

Manual Search Mode

The *Manual Search Mode* requires to know where the second client is located. After the transmission of the parameters for a desired scenario using the *Manual Search Mode*, the *uavcontroller* on the wireless mesh node responds that it awaits a *sendFlightDirection* message. The estimated position of the second client can be set by touching on the map of the *Remote Control App*. After the UAV has received the message, it sets the contained coordinates as the next waypoint and flies towards its direction.

Autonomous Search Mode

The *Autonomous Search Mode* does not require a *sendFlightDirection* message, in contrast to the *Manual Search Mode*. If the *uavcontroller* receives a *submitStartConfiguration* message that contains the *Autonomous Search Mode*, it will calculate multiple waypoints forming a spiral around its location. Then, the UAV will begin to follow the flight path for the detection of the second client.

Airborne Relay

The *Airborne Relay* network scenario uses one UAV as carrier for the WMN and requires two clients for a successful deployment. Because the clients are not too far away from each other, it is sufficient to use only one UAV for the connection. The *Airborne Relay* scenario is illustrated in Figure 2.10 and its start configuration supports the following modes:

- Location Position Mode
- Signal Strength Position Mode
- Manual Search Mode
- Autonomous Search Mode

In this scenario, the UAV starts within the signal range of the first client and the managing device running the *Remote Control App*. The UAV does periodically broadcast *ping* messages. When the first client receives such a message, it answers with the *submitPosition* message. Then the UAV knows the GPS coordinates of the first client. After the UAV received the *submitStart-Configuration* message from the *Remote Control App*, the UAV starts the deployment depending on the submitted parameters. If the *Manual Search Mode* was chosen, it requests the *sendFlight-Direction* message. Otherwise, it instantly starts the deployment using the *Autonomous Search Mode*.

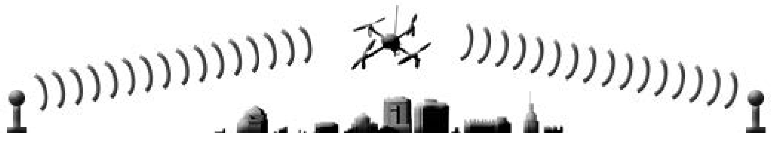


Figure 2.10: *Airborne Relay* scenario

The UAV begins to fly towards the submitted waypoint or on a flight path forming a spiral around its location, depending on the search mode. After a while, the UAV gets into the transmission range of the second client. If the second client receives the broadcasted *ping* message, it answers with the *submitPosition* message. From this moment on, the UAV knows both locations of the clients. Therefore, it flies to the center of their locations according to the *Location Position Mode*. If the start configuration contained the *Signal Strength Position Mode*, the UAV additionally changes its position to provide equal signal strength in reference to both clients. When the UAV comes back into the signal range of the managing device, the position of the second client is known by the *Remote Control App* as it receives this information through the *notification* messages. Both clients can communicate with each other over the WMN when the UAV reached its final position.

To finish the *Airborne Relay* scenario the *sendAbort* message has to be submitted. Afterwards, the UAV will return to its start location.

Airborne Multihop Relay

The *Airborne Multihop Relay*, as illustrated in Figure 2.11, uses multiple UAVs and forms a communication chain between two clients that provides multihop network transmission between them. If one UAV is not enough to establish a WMN that covers both clients, this scenario can provide a solution. It supports the *Manual Search Mode* and the *Autonomous Search Mode*. The positioning behavior is not part of its start configuration since it uses always the same positioning procedure. Additionally, this scenario requires that the UAVs are placed on different altitudes because they do not have a collision avoidance system.

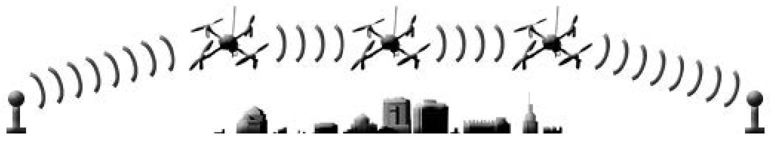


Figure 2.11: *Airborne Multihop Relay* scenario

The UAV that receives the *submitStartConfiguration* message from the *Remote Control App* starts the deployment process. Similar to the *Airborne Relay* scenario, the UAVs receive the *submitPosition* message from the first client. Afterwards, the starting UAV begins to fly depending on the search mode.

When the UAV detects the second client, it includes the information about its position into the *ping* and *notification* messages. As a consequence, the remaining UAVs get informed about the position of the second client, as soon as the UAV comes back into their signal range. After the detection of the second client, the first UAV flies towards the geographical center between the clients according to the *Location Position Mode*.

When the center position is reached, the UAV begins to fly towards the first client. If the connection to the first client achieves a certain predefined signal strength, the UAV stops its movement. Then the UAV propagates that it reached its final position over the *ping* messages.

Every time when a UAV has reached its final position the following UAV flies towards its location. Afterwards, it flies towards the direction of the second client until the signal strength between itself and the last deployed UAV reaches a certain value. This procedure is repeated until all UAVs are placed accordingly.

To end the *Airborne Multihop Relay* scenario the *sendAbort* message has to be submitted to a participating UAV. Afterwards, the UAVs will return to their start location.

Network Area Coverage

The *Network Area Coverage* scenario is intended to provide network coverage for a certain polygonal area using multiple UAVs. After submitting the coordinates of the area as a part of the start configuration, *UAVNet* positions the UAVs autonomously within the area so that their positions provide the best possible network coverage. This kind of scenario requires a collision avoidance system, a smart positioning mode, as well as swarm behavior of the UAVs. Although, this mode is not yet implemented in the prototype of *UAVNet*, some of its future requirements have influenced the development of *UAVNet* and the *Remote Control App*. For this scenario, the *Remote Control App* includes the functionality to draw a polygonal area on the map, but it does not provide any further processing.

2.1.5 UAV Simulator

The *uavcontroller* is able to run as a simulator for testing and developing of managing and monitoring applications like the *Remote Control App*. It uses a predefined list of comma-separated values that are sent with the *ping* and *notifiction* messages. The simulator only handles *sendNotificationSubscription* messages. It simulates a UAV notification service and its broadcasted *ping* messages. Other types of message transmissions are not supported by the simulator, such as the response to *sendFlightDirection*, *submitStartConfiguration* and *sendAbort* messages. The response values have to be hard-coded in the application for testing and debugging. The transmission of a start configuration for a *UAVNet* scenario does not affect the values sent by the simulator. An example list of the comma-separated values is included in the *UAVNet* software. As shown in Listing 2.1, the simulator mode is initialized by adding the *-s* flag to the start command of the *uavcontroller*.

```
$ ./uavcontroller -s
```

Listing 2.1: Starting *uavcontroller* in the simulation mode.[1]

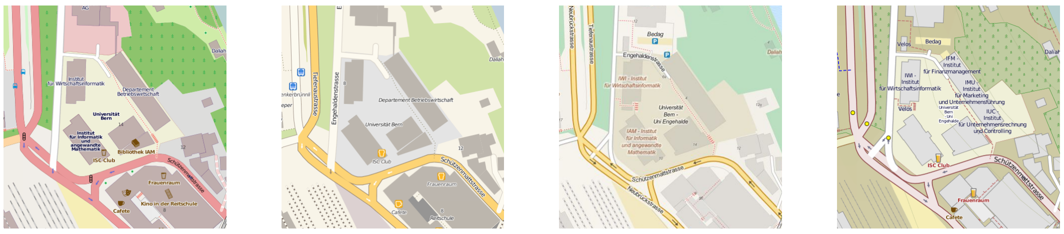
2.2 Route-Me

Route-Me[12] is an open-source Objective-C library for displaying maps on iOS devices. The library depends on the PROJ.4 cartographic projections library[13]. It was initially developed by Joseph Gentle as the basis for a transit routing application[14]. Gentle's application was never completed due to government licensing issues. As a consequence, the *Route-Me* library was released in September 2008 as open-source under the New BSD license[15] and is currently hosted on GitHub[12].

The *Route-Me* library provides several advantages for map representations compared to other frameworks. It is capable to use locally stored, precompiled databases containing map tile images and to integrate different online map data sources. Additionally, the library provides a configurable caching mechanism that creates databases of tile images from online map sources.

2.2.1 Map Sources

Since different users have different preferences, *Route-Me* provides integration for many various map sources including OpenStreetMap (OSM), CloudMade, MapQuest and OpenCycleMap, shown in Figure 2.12.



(a) OSM standard tile image (b) CloudMade tile image (c) MapQuest tile image (d) OpenCycleMap tile image

Figure 2.12: Different map sources based on OSM data

These sources used by the *Remote Control App* are described as follows:

OpenStreetMap
OSM is a collaborative project of volunteers, who collect mapping information all over the world, to provide a free and editable map. The map from OSM is ”Open Data”. This means that everyone can utilize the data for his own usage as long as its source is mentioned. All tile images from OSM are 256 x 256 pixel PNG files.

CloudMade
CloudMade is a company, which provides commercial applications with map data. CloudMade’s maps are based on OSM data and provide many different map styles using various coloring and different map labels. The usage of CloudMade maps requires an API-key.

MapQuest
MapQuest is currently the second-largest mapping service company according to their owner AOL[16]. It provides desktop, mobile and business solutions, and uses map data from both commercial and free sources. The data from free sources is largely based on OSM data.

OpenCycleMap

OpenCycleMap is a OSM rendering layer and is primary focused to show relevant information for cyclists.

2.2.2 Cache

The default cache structure of the *Route-Me* library consists of a memory cache of 32 tile images, followed by a SQLite-based database cache that is created in the document folder of the *Remote Control App* at the first use of each online source[17]. The database cache stores by default the 1000 most recently downloaded tile images. The library creates for each tile source a separate database file that is stored by default in the application's documents folder. The cache strategy for the database cache is by default Least Recently Used (LRU). It is possible to configure the *Route-Me* library to use First In First Out (FIFO). This is a more performant purge strategy since LRU needs a write process for marking the date after each read access, while FIFO does not.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.
    apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>caches</key>
        <array>
                <dict>
                        <key>type</key>
                        <string>memory-cache</string>
                        <key>capacity</key>
                        <integer>48</integer>
                </dict>
                <dict>
                        <key>type</key>
                        <string>db-cache</string>
                        <key>capacity</key>
                        <integer>1024</integer>
                        <key>strategy</key>
                        <string>FIFO</string>
                        <key>useCachesDirectory</key>
                        <true/>
                        <key>minimalPurge</key>
                        <integer>10</integer>
                </dict>
        </array>
</dict>
</plist>
```

Listing 2.2: Example of a cache property list[17]

To adapt the default cache configuration, a custom property list termed "routeme.plist" can be created and included as target to the application build. The property list is a dictionary with the key *caches* which contains an array of dictionaries defining the parameters for each cache type. Every entry in the *caches* array must contain the key *type* indicating either memory or database cache by values *memory-cache* and *db-cache*. The supported parameters are listed in Table 2.1 and an example is shown in Listing 2.2. If the property list misses an entry for the database cache, it will not instantiated.

Parameter	Type	Description
capacity	Number	The number of tile images kept by cache. No upper limit can be set by 0 value.
strategy	String	The purge strategy has to be set to either LRU or FIFO. (database cache only)
useCachesDirectory	Boolean	Use the App Caches folder or App Documents folder. (database cache only)
minimalPurge	Number	The smallest amount of images removed during a purge operation. Must be at least one and at most the cache capacity. (database cache only)

Table 2.1: Supported parameters for cache types in routeme.plist

2.2.3 Offline Maps

Since the *Remote Control App* is supposed to be used in an enviroment without Internet access, the opportunity of using an offline source for the map representation in the application is significant. The *Route-Me* library has the functionality to use two different database-backed formats. These are called *DBMap* and *MBTiles*. The *Remote Control App* uses the *DBMap* format for offline map sources. The *DBMap* format is a generic database file of the db-format containing the map tiles of a certain area. The download of tile images and the following creation of the database file cannot be accomplished by a mobile device itself in an usable way because of the long-term download time. Depending on the size of the area and the supported zoom level, the database file can grow quickly to a large amount of data even for small areas.

For the generation of a map tile database, the custom open-source PERL script called ”downloadosmtiles.pl”[18] and the command line tool ”map2sqlite”[19] are needed. The downloadosmtiles.pl script provides the functionality to download the map tile images of a certain area from various sources. Its parameters are shown in Table 2.2. The square brackets stand for optional values. A missing maximum value for latitude or longitude will lead to download only tile images affected by the minimum value, like tile images laid over the minimum latitude or longitude. The Listing 2.3 shows an example command to download map tiles.

```
$ downloadosmtiles.pl --lat=46.95173:46.95558 --long
    =7.43745:7.44288 --zoom=12:15 --destdir=/tiles/folder
```

Listing 2.3: Example of a command to download map tiles.

The Table 2.3 shows the various sources that can be used as value for the *baseurl* parameter including its supported zoom levels[21]. The square brackets stand for alternative

Parameter	Syntax	Description
latitude	min[:max]	Minimum and maximum latitude of the bounding box of coordinates to download
longitude	min[:max]	Minimum and maximum longitude of the bounding box of coordinates to download
zoom	min[:max]	Minimum and maximum zoom level
link	URL	A URL consisted of tile source, latitude, longitude and zoom level information
baseurl	URL	The base URL of the download server
destdir	directory	The destination for the downloaded tile on the local file system
quiet		Switches off the verbose mode
dumptilelist	filename	Writes a file in YAML format[20] containing the list of the tiles. The script will not download map tile images when using this parameter.
loadtilelist	filename	Does download the tiles from a given list.

Table 2.2: Supported parameters for downloadosmtiles.pl[18]

values because the tile sources have multiple subdomains to download the tiles from (e.g. a.tile, b.tile, c.tile), and the Listing 2.4 shows the command for the creation of the generic database file. The folder structure of the downloaded map tiles is shown in Figure 2.13 where the top level folder names denote the zoom level. The second level folder names denote the value of the X-axis and the name of the tile images stand for the value of the Y-axis, both based on the Mercator projection[21]. The resulting folder structure has the shape */zoom/x/y.png* and is used also as URL template by all supported download sources (e.g. http://a.tile.openstreetmap.org/zoom/x/y.png).

Source	URL	Zoom Levels
OpenStreetMap	http://[abc].tile.openstreetmap.org	0-19
MapQuest	http://otile[1234].mqcdn.com/tiles/1.0.0/map	0-19
MapQuest Open Aerial	http://otile[1234].mqcdn.com/tiles/1.0.0/sat	0-11 (global), 12+ for U.S.
OpenCycleMap	http://[abc].tile.opencyclemap.org/cycle	0-18
OpenCycleMap Transport	http://[abc].tile2.opencyclemap.org/transport	0-18
CloudMade	http://[abc].tile.cloudmade.com/your-API-key/1/256	0-18

Table 2.3: Examples of sources for map tile images[21]

```
$ ./map2sqlite -db mapname.sqlite -mapdir /tiles/folder
```

Listing 2.4: Command that creates the SQLite database file from a tiles folder.

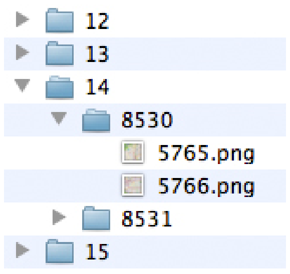


Figure 2.13: Structure of downloaded tile images

Chapter 3

Remote Control Application

As front-end application for the *UAVNet*, the *Remote Control App* has been developed. The application provides a comfortable interface to manage the *UAVNet* system. It provides the functionality to set up and to monitor *UAVNet* and to abort the deployment procedure of the UAVs. It stores the information about past deployments in the application's underlying SQLite database as well as the *notification* messages sent by the UAVs during a flight scenario. It is implemented for the iOS operating system as a so-called ”universal app”, which means, that it is both running on iPhone, iPod and iPad devices with the same code basis. Furthermore, the *Remote Control App* running on iPad adjusts its graphical user interface automatically, depending on horizontal or vertical device orientation. The *Remote Control App* is a collaborative project with [1] and [22]. The application functionalities are:

- Configuration, setup and deployment of one to several UAVs for the supported network scenarios.
- Geographical representation of the device, clients and UAVs on a map including graphical representation of actual sensor values transmitted by the UAVs.
- Integration of various online and offline map sources.
- Monitoring of an already deployed *UAVNet*.
- Storage of the received messages.
- Reviewing of past flights.

This chapter discusses the architecture of applications following the Model-View-Controller (MVC) design pattern that is used by the *Remote Control App* as well as any other iOS application in Section 3.1, followed by an overview about the implementation of the *Remote Control App* including a short description about the Objective-C language and the construction of its database in Section 3.2. As the *Remote Control App* consists of many different views, Section 3.3 provides the presentation of the graphical user interface (GUI) in detail. The message flow between the *Remote Control App* and the *UAVNet* is investigated in Section 3.4. The Section 3.5 concludes this chapter as it declares the setup procedure with *Remote Control App* for a network scenario.

3.1 Architecture

This section is about the architecture of the *Remote Control App*. Since every iOS application should follow the Model-View-Controller (MVC) design pattern and as the provided frameworks built into the iOS operating system correspond to this concept, it is described first and has to be known to understand the construction of those applications.

3.1.1 Model View Controller Design Pattern in iOS

The MVC design pattern splits up the parts of an application into three interconnected components, each responsible for different processings and each specialized for its task[4]. These three different roles are termed *model*, *view* and *controller*, and have to interact with each other. The advantage of this type of architecture is the convenience with regard to maintenance and reusing of the programmed parts. This concept is illustrated in Figure 3.1.

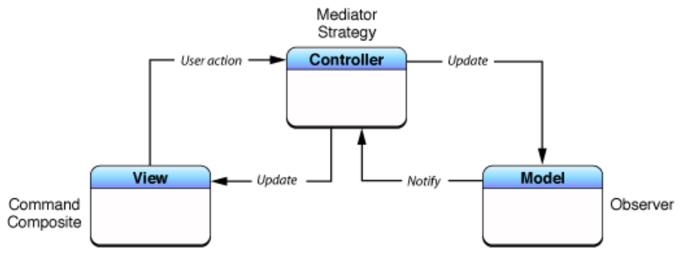


Figure 3.1: Cocoa version of MVC as a compound design pattern[23]

The *model* contains the data to be presented or processed. The *view* is responsible for the interaction with the user as well as for the presentation of the data. The *controller* handles the flow of data between the *view* and *model*, thus it acts as *mediator* between them. The changes of the data in the *model*, such as updated database rows or changed object values, are notified and processed by the *controller*, and afterwards delegated to the *view* if this is necessary. Conversely, the *view* takes care of user input and delegates the user action to the *controller* where the information is processed and transfered to the *model* once its necessary. For the Objective-C programming language the different parts can be described as follows:

- A *model* is typically a subclass of NSObject, which is the root class for most inheritances of Objective-C classes, or a database data type like those provided by a Core Data model as described in Subsection 3.2.3.
- A *view* is usually a UIView object or a XIB file that acts as a container for multiple different classes and subclasses of UIView.
- A *controller* is typically a class or subclass of UIViewController that acts as *mediator* between its associated *models* and *view*.

3.2 Implementation

The current implementation of the *Remote Control App* supports the *Airborne Relay* and *Airborne Multihop Relay* modes of the *UAVNet*. It additionally provides a *Monitor Mode*, which subscribes to the UAVs managed by an other device. The *Monitor Mode* can be used by other users to gather information about the current state of a *UAVNet* deployment without the need to set up and to control the deployment process. The different views are implemented as XIB files. A XIB file is a XML representation of an object graph of various classes and subclasses of UIViews that defines the structure of a certain view in the application.

Almost the entire code is written in Objective-C, exept for some small C code parts that are needed to handle the *ping* and *notification* messages of *UAVNet*. The threads that are responsible for listening to *ping* and *notifiaction* messages are placed in their responsible *controller* class. Each received *notification* message gets processed by the observing thread into the database. Multiple *controller* classes track database changes and trigger the representation update of UAVs and clients on the *map* view or the detailed list of the UAVs current values on their corresponding table view. The application uses the default cache behavior of the *Route-Me* library, as described in Subsection 2.2.2, and it integrates common touch-based user actions, such as the swiping over table cells for deletion.

The application consists of the *documents*, *library* and *tmp* folders on the file system level as any other iOS application since these folders are part of the sandbox environment, but only the *documents* folder is used directly by the *Remote Control App* for storage of the *Route-Me* cache and the primary databases. As the contents of the *documents* folder are accessible over the iTunes software, this provides the possibility to export and to backup the complete primary database of the *Remote Control App* as well as to delete unwanted cache databases. To include an offline source for the representation of the map, the tile database simply has to be placed in the folder. Additionally, the contents of this folder are backed up during the synchronisation procedure between the iOS device and iTunes.

An other aspect is the behavior of the *Remote Control App* when its execution is interrupted. As an example, an incoming phone call during the deployment of *UAVNet* forces the application into the ”background” state. In this state, the application has only a few seconds to perform algorithms to ensure that it will not crash because after a few seconds the application will enter the ”suspended” state where no more code execution is possible. Therefore, the *Remote Control App* assures that all cached changes of the database, which are currently not saved to the SQLite database file and are only stored in the main memory, are synchronized with the SQLite database file when entering the ”background” state. iOS applications are forwarded to the ”suspended” state shortly after entering the ”background” state. The suspended applications remain in the memory and cannot execute any code. An other problem, which comes with the ”suspended” state, is reclaiming of resources. This could cause that a socket used by the *Remote Control App* gets closed. Although, the application ensures that the closed sockets are reestablished when entering back into the ”active” state, some messages from the *UAVNet* are irrevocably lost.

3.2.1 Implementation of the Remote Control App

The impementation of the *Remote Control App* follows the MVC design pattern. The application includes the *Route-Me* and *libuavext* libraries. The *model* of the application is extended by several exportable databases. There is a cache database for each online map source that is managed over the *Route-Me* library. An additional primary database, managed by several *controller* classes, is used for the storage of received messages, past *UAVNet* deployments and the application stettings. Additionally, the *controllers* manage the display of their associated *view* according to the current state of the application. The application provides multiple *views* that are described in Section 3.3. By using the *libuavext* library, it is able to listen to *ping* and *notification* messages from *UAVNet* as well as to submit *control* messages using the IEEE 802.11g standard. The implementation of the *Remote Control App* is illustrated in Figure 3.2.

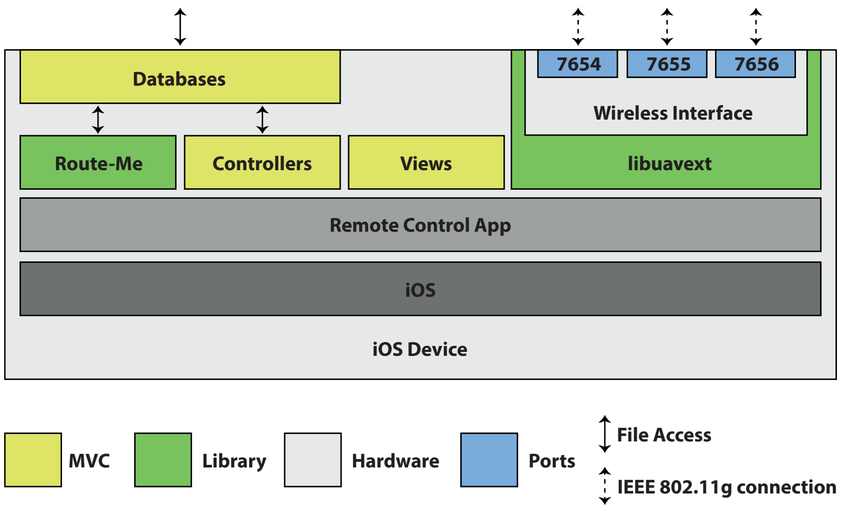


Figure 3.2: Scheme of the implementation of the *Remote Control App*

The *Remote Control App* uses several frameworks that are provided by the iOS layers as previously seen in Figure 1.1. These are the Foundation, UIKit, Core Data and Core Location frameworks. The following paragraphs give a short overview about these frameworks.

Foundation Framework

According to its term, the Foundation framework provides the base layer for Objective-C classes[4]. It is part of the Core Services layer and provides the root class NSObject, which is the basis of many widely used and inherited subclasses. Further, it provides so-called ”utility classes” and introduces consistent conventions. Its portable code enhances the portability of other frameworks that build on the Foundation framework and implementations towards other platforms.

UIKit Framework

The UIKit framework contains all needed classes for the GUI-development of the iOS operating system and it is specially optimised for touch-based input. Therefore, it is part of the Cocoa Touch layer[4]. It provides an application object, event handling, drawing model, windows, views and controls.

Core Data Framework

The Core Data framework is used to manage relational entity-attribute models and data, such as SQLite databases and XML[4]. It is a part of the Core Services layer providing relationship maintenance, tracking and undo support, validation of attribute values, observing, merge policies, search query compilation and exportable data. As it is essential to handle the primary database, its functionality has been included into the *Remote Control App*.

Core Location Framework

The Core Location framework is provided by the Core Services layer[4]. It is used to access the hardware providing the device's geographical location or orientation, thereby this framework is needed by the *Remote Control App* to find its own location and to adjust the orientation of the map towards the magnetic North direction.

3.2.2 Objective-C

Objective-C is the primary programming language used to write software for iOS as well as for Mac OS X. The concept of the syntax of Objective-C is based on the Smalltalk programming language. It extends the procedural C programming language with object-oriented concepts and forms a strict superset of C. Therefore, every C-program or C-library, such as the *libuavext* library, can be compiled through an Objective-C compiler and C code can be included directly within an Objective-C class.

3.2.3 Database

The interaction with the primary database is implemented with the services provided by the Core Data framework. The *controllers* use observing methods to track changes of the database objects and adjust the views according to the MVC concept. On the other side, user generated input, such as the changing of settings over the GUI, is processed by the *controllers* and the information is forwarded to the database if this is necessary. The model of the database is illustrated in Figure 3.3.

There are several objects with different relationships among each other. These are the *SettingState*, *MacAddress*, *Flight*, *UAV*, *Station*, *Notification*, *Status* and *Location* objects and are described as follows:

SettingState
The *SettingState* object is responsible to contain the current settings of the *Remote Control App*. These settings include the API key for map sources provided by the CloudMade

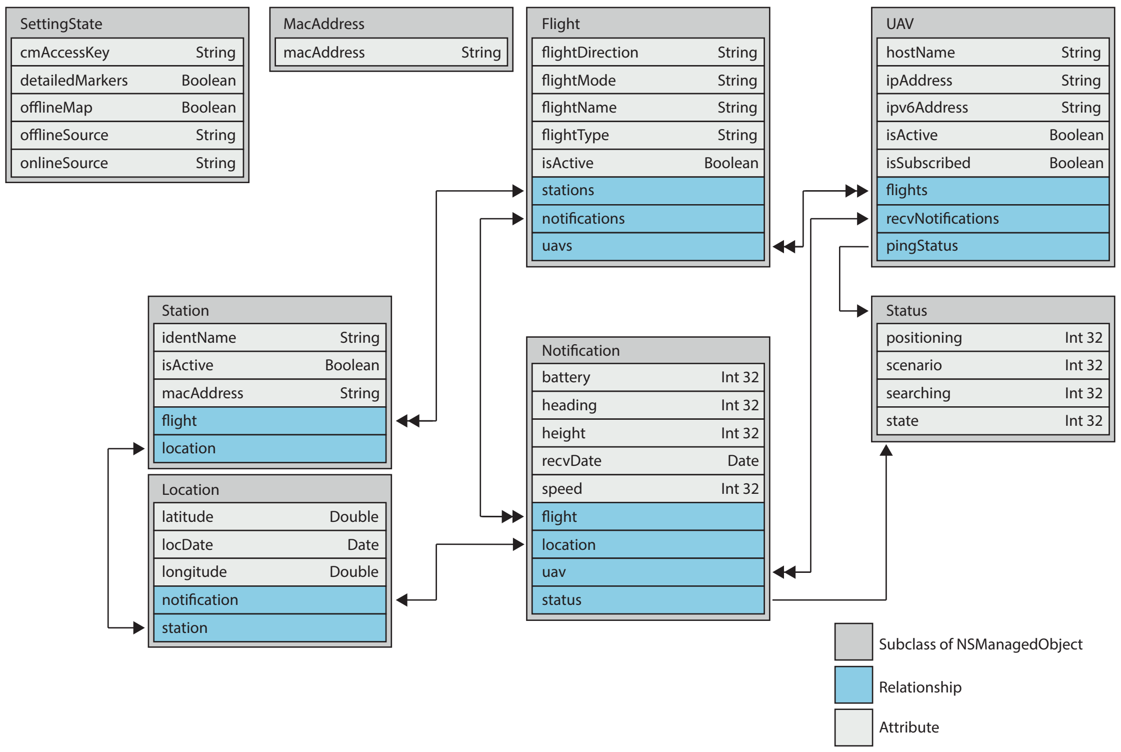


Figure 3.3: Core Data Model of *Remote Control App*

company, the filename of the chosen offline map database and the name of the online source as strings. Additionally, a boolean value indicates whether to use an offline map or online map source. An other boolean value is used to switch between the use of detailed markers for the representation of the UAVs and the simple UAV representation, as described in Subsection 3.3.1. There can be only one *SettingState* object in the database at any time.

MacAddress

There can be several *MacAddress* objects in the database. These are used to manage the allowed MAC addresses of the clients of the *UAVNet*. As it is very uncomfortable to type in MAC addresses over a touch-based GUI, they can be stored permanently. The list of the allowed MAC addresses can be managed over the *options* view as explained in Subsection 3.3.5.

Flight

A *Flight* object is created when the user sets up the parameters of a desired *UAVNet* scenario. Each *Flight* object stores the information about a completed, past *Airborne Relay*, *Airborne Multihop Relay* or *Monitor Mode* scenario. It contains the type of the network scenario, the type of the position and search mode, as well as a name for it. The name is intended to give additional information for the management over the *archive* view as illustrated in Figure 3.7. Additionally, a boolean value flags a *Flight* object if it is currently active. This would indicate that there is an ongoing *UAVNet* deployment or a flight scenario is set. The *Flight* object contains two one-to-many relationships to determine the client notebooks as *Station* objects as well as all notifications that have been received during this flight scenario. Additionally, there is a many-to-many relationship between the *Flight* objects and the *UAV* objects as a flight can have multiple associated UAVs and a UAV can be part of several past flights.

UAV

A *UAV* object is created as soon as an unknown UAV gets discovered for the first time. It contains the hostname of its associated UAV as an unique identifier, its IPv4 and IPv6 address, a boolean value to indicate if the *UAV* object is marked as active and an additional boolean value to point if the device is currently subscribed to the UAV's notification service. A *UAV* object which is marked as active indicates that its associated UAV has been recently detected through listening for *ping* messages. After a *UAVNet* deployment is finished, the *UAV* objects are marked as inactive. Since the UAVs associated with the *UAV* objects can be part of multiple past *UAVNet* deployments and because a *UAVNet* deployment can consist of several UAVs, a many-to-many relationship is used between the *Flight* and *UAV* objects. Additionally, all notifications that have ever been received by the UAV are associated with a one-to-many relationship. The *pingStatus* attribute is referencing to a single *Status* object. It contains the values of the last received *ping* message and is updated every time when the *Remote Control App* listens to these messages for the detection of currently unknown UAVs.

Station

The *Station* object is used for the clients of *UAVNet*. Similar to the *Flight* and *UAV* objects, a boolean value indicates if the client is part of an active scenario. The other stored values are the name of the client that either can be Client1 or Client2, as well as its MAC address. It provides an one-to-one relationship to indicate the position of the client by a *Location* object. An one-to-many relationship is used between the *Flight* object and its associated clients.

Status

The *Status* object stores the information contained in the *ping* messages as well as the contained *Status_t* structure in the *notification* messages. A single *Status* object is referenced by either a *Notification* object or a *UAV* object.

Notification

For each received *notification* message of the *UAVNet*, a *Notification* object is created. It consists of the information contained in the message indicating the current state of a UAV. That includes a reference to a single *Status* object as the *Status_t* structure is part of a *notification* message. A one-to-one relationship is used to associate the contained location information to a *Location* object. Additionally, two one-to-many relationships are used to associate *Notification* objects to the sending UAV and to the associated flight.

Location

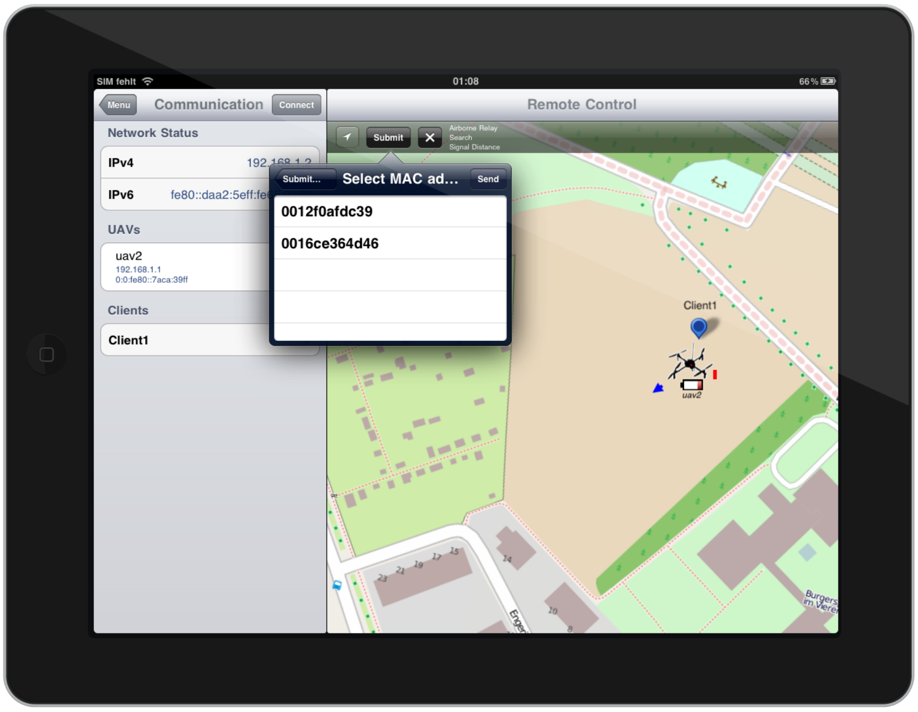
The *Location* object consists of the longitude, latitude and date for a certain GPS position. It can either be connected to a *Station* or *Notification* object over a one-to-one relationship.

3.3 Graphical User Interface

The *Remote Control Application* is composed of the *map*, *scenario*, *communication*, *archive* and *options* views. To switch between the views on the iPhone or iPod, the navigation is accomplished through a tab bar on the bottom. For the iPad, the navigation is shown as drop down menu for a vertical device orientation. An horizontal device orientation leads to a splitted view consisting of the map view on the right side and the remaining views on the left side.

3.3.1 Map View

A purpose of the *map* view is the representing of the clients of the *UAVNet*, the current geographical position and states of subscribed UAVs, and the location of the mobile device on the map. The location of the mobile device is marked by a blue point and the detected clients are shown using a blue marker with either the label "Client1" or "Client2". The UAV representation can be a detailed marker or a simple red point. Figure 3.4 shows the map view as seen on an iPad device.



Network Status

IPv4 192.168.1.2

IPv6 fe80::daa2:5eff:fe

UAVs

uav2
192.168.1.1
0:0:fe80::7aca:39ff

Clients

Client1

0012f0afdc39

0016ce364d46

Figure 3.4: *map* view with detailed UAV marker representation on iPad

The layout of the detailed marker is shown in Figure 3.5. For the detailed view of a UAV, the representation includes an arrow indicating the current flight direction and speed by orientation, length and coloration. This means that the faster the UAV, the longer the arrow is on the *map* view. The exact position of the UAV is shown by a red dot on the UAV graphic. An adaptive colored bar is placed on the right side of the marker that shows the altitude over ground in relation of the take-off height. On the bottom of the detailed marker, the battery gauge and the hostname are shown. The coloration of the direction and speed vector, battery gauge and altitude bar is either red, yellow or blue. The blue color refers to good states. This decision was made to give a better contrast since there might be large green areas on the *map* view. The red color stands for dangerous states as low altitude, high velocity and low battery charge. The yellow color indicates acceptable states.

Because the *Manual Search Mode* requires a waypoint location, the user can set such a location by touching on the map. When a location is set, it is shown with a red direction marker similar to the client marker. Additionally, a line is drawn between the UAV and the direction marker that indicates the direction on the map. As soon as the direction marker is set, it can be moved to adjust the desired position.

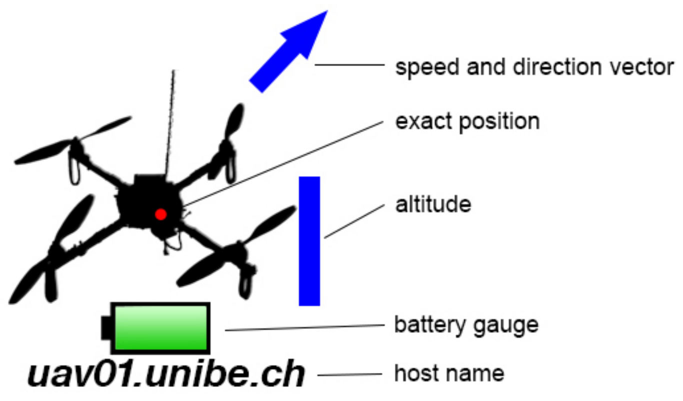


Figure 3.5: GUI-Marker showing the current state of a UAV

Control Bar

An other purpose of the *map* view is to manage the current flight scenario. For this purpose, it holds a control bar on the top. The control bar consists of a *location*, a *state* and a *cancel* button from the left to the right. Additionally, a field on the right side displays the type of the current flight scenario.

Location Button

Pushing the *location* button one-time will execute the location tracking of the mobile device by setting and moving a blue point at the device's GPS coordinates on the map. The *map* view will be centered at the blue location point during location tracking. A second push on the *location* button will additionally perform automatic rotation of the map towards the magnetic North direction using the integrated digital compass. Location tracking can be aborted by swiping the map or pushing the *location* button again.

State Button

The *state* button allows to forward to the next state of a flight scenario and changes its label according to the current state of the *Remote Control App*. The procedure to manage a network scenario is described in Section 3.5.

Cancel Button

The *cancel* button has multiple purposes, depending on the current state of the *Remote Control App*. It can be used to cancel a network scenario before the *submitStartConfiguration* message was sent to the UAVs. Since the UAVs are flying the network scenario after receiving this message, the *sendAbort* message has to be submitted to end the deployment process. This procedure is provided by the *state* button. Another purpose is to remove drawn flight paths provided by the *archive* view.

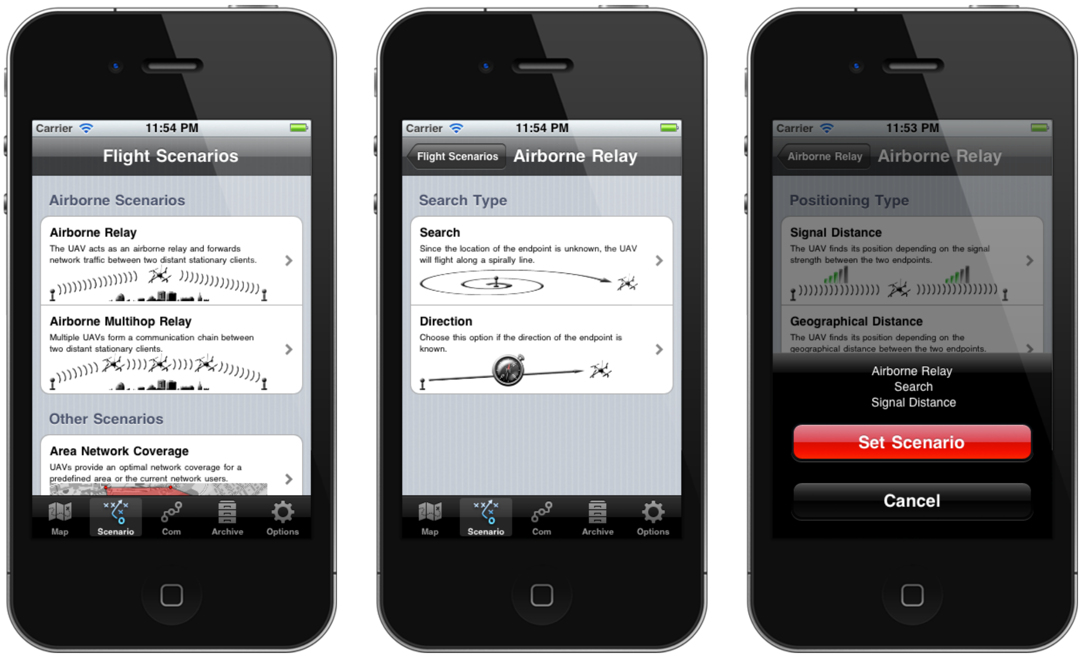
3.3.2 Scenario View

The *scenario* view simply provides the ability to set up the parameters of a desired network deployment for the *UAVNet*. First, the type of the scenario has the be chosen. This either can be an *Airborne Relay*, *Airborne Multihop Relay* or *Monitor Mode* according to 2.1.4. There is a selection for *Network Area Coverage*, but it is disabled since this scenario is currently not supported. The selection of this scenario would forward to a drawing mode for the polygonal area. After chosing the type of the scenario, the next view slides in, requesting additional setup information depending on the chosen scenario. As last step, the user is asked to confirm his selection. Once a scenario is set, the user cannot set up an other scenario as long as the initial scenario is active. The Figure 3.6 illustrates this procedure for the *Airborne Relay* scenario.

3.3.3 Communication View

The *communication* view is shown in Figure 3.4 next to the *map* view. It provides a *connect* button on the top left and consists of a *Network Status*, an *UAVs* and a *Clients* section. The *Network Status* section simply shows the IPv4 and IPv6 address of the device. A purpose of the *connect* button is to listen to the broadcasted *ping* messages by *UAVNet*. For the subscription to the notification service and the transmission of the start configuration of a network scenario, the participating UAVs have to be discovered previously. Once the *Remote Control App* has received such a message from a UAV, it is shown under the *UAVs* section. Accordingly, the *clients* section gets updated if clients within the *UAVNet* are discovered.

Furthermore, the selection of a client row will forward to a table view where the actual values of its status is shown. The selection of a UAV does the same, but the information will also include the values of the last received *notification* message. If the device is not subscribed to the selected UAV, it will show only the information of the last received *ping* message.



The UAV acts as an airborne relay and forwards network traffic between two distant stationary clients.

Multiple UAVs form a communication chain between two distant stationary clients.

UAVs provide an optimal network coverage for a predefined area or the current network users.

Since the location of the endpoint is unknown, the UAV will flight along a spirally line.

Choose this option if the direction of the endpoint is known.

The UAV finds its position depending on the signal strength between the two endpoints.

The UAV finds its position depending on the geographical distance between the two endpoints.

(a) Selection of a scenario

(b) Selection of the search mode

(c) Selection of the position mode with confirmation

Figure 3.6: *scenario* view of the setup procedure for *Airborne Relay* on iPhone

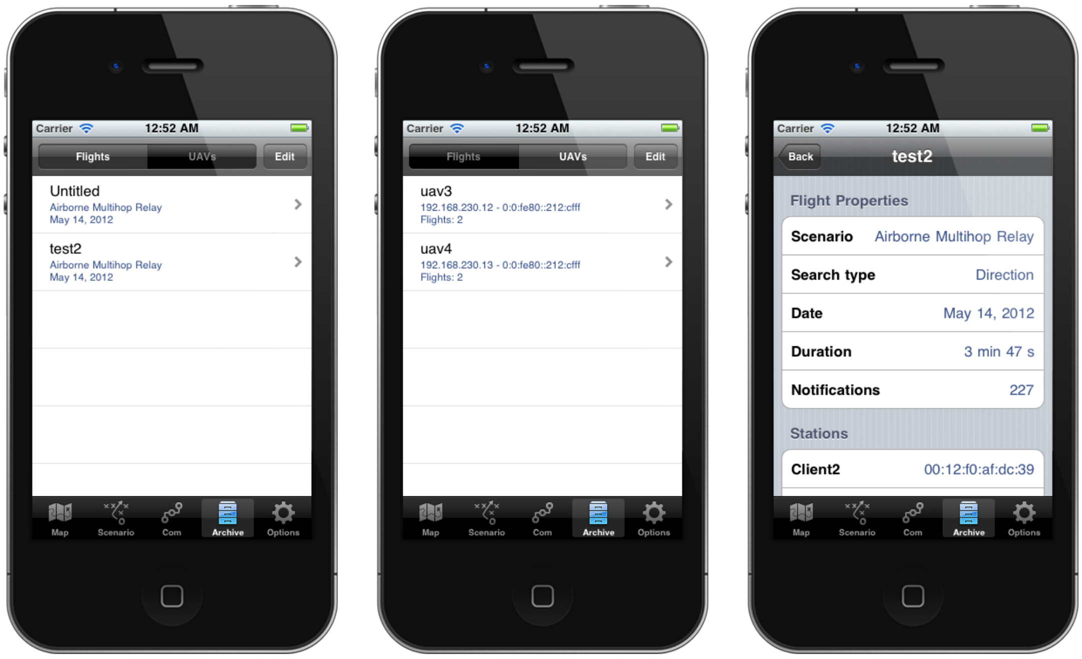
3.3.4 Archive View

The *archive* view displays information about the saved data from past network scenarios. The different parts of the *archive* view are shown in Figure 3.7. The view uses the information contained by the stored *notification* messages. It is separated into two sections. On the top, a tab bar is placed where the user can choose between a *Flights* or *UAVs* overview.

The *Flights* selection presents all locally stored flights, one for each row. The selection of a flight forwards to a table view that presents stored and calculated information, such as the duration of the network deployment by calculating the time between the first and last received *notification* messages that are associated with this flight. Furthermore, the date, the search and position mode, the number of received messages, the MAC addresses of the found clients, as well as the hostnames of the participated UAVs are shown. Additionally, the option to draw the flight paths of the UAVs onto the map is available at the bottom.

The *UAVs* selection lists all stored UAVs including the number of participated flights. The selection of a UAV will forward to a table view where each associated flight of the selected UAV is shown similarly to the *Flights* selection. Then, the selection of a row forwards to the table view that presents the detailed information about the flight.

If a scenario is active, the flight paths cannot be drawn on the *map* view. This prevents confusion since the labels and the area of the scenario might be similar. Additionally, the data of the UAVs as well as the saved flights can be removed by using the edit button on the top.



Untitled
Airborne Multihop Relay
May 14, 2012

test2
Airborne Multihop Relay
May 14, 2012

uav3
192.168.230.12 - 0:0:fe80::212:cfff
Flights: 2

uav4
192.168.230.13 - 0:0:fe80::212:cfff
Flights: 2

Flight Properties

Scenario	Airborne Multihop Relay
Search type	Direction
Date	May 14, 2012
Duration	3 min 47 s
Notifications	227

Stations

Client2	00:12:f0:af:dc:39

(a) *Flights* part of the archive (b) *UAVs* part of the archive (c) Detailed view about a flight

Figure 3.7: Different parts of the *archive* view on iPhone

3.3.5 Options View

The options for the *Remote Control App* are separated into the *database*, *general* and *tile source* section. The *database* section provides options to manage the content of the tile image databases as well as the primary database of the *Remote Control App*. It is followed by the *general* settings that provide common options for the representation of the map as well as the MAC address configuration for the network scenario of the *UAVNet*. The *database* section provides the selection of the map sources. The *options* view as displayed for an iPhone is shown in Figure 3.8. The individual settings are described as follows:

Clear image cache
: The user can clear out all tile images from the *Route-Me* database cache of the current map source with the selection of this item.

Clear NSURL cache
: The different online tile source implementations load tile images via NSURLRequest.

There is a possibility that some tile images will remain in the applications shared URL cache. If the user wants to clear them out, he can use this option.

Clear database
This option clears out all past flights including all information associated with them. This option does not affect the allowed MAC addresses list as well as the current flight scenario and UAVs, which are marked as active.

Offline Map
Using the provided switch, the user can enable or disable the use of an offline tile source.

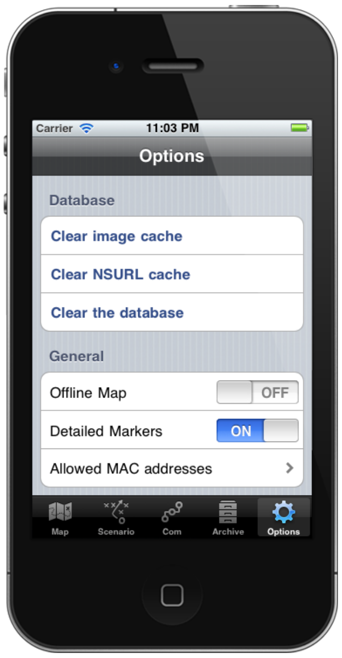
Detailed Markers
This switch enables or disables the use of detailed markers for the UAV representation.

Allowed MAC Addresses
This item forwards to a table view where the user can modify the list of allowed MAC addresses for the *UAVNet*.

Online Source
This item opens a view where the user can choose between several predefined online map sources.

Offline Source
This option forwards to a table view where the user can choose between all locally stored offline map databases.

Cloud Made API Key
The usage of CloudMade maps requires an API-key. This item forwards to a view where the user can type it in.



Database

Clear image cache

Clear NSURL cache

Clear the database

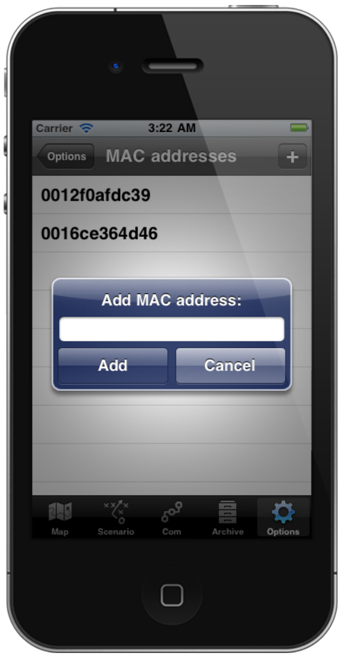
General

Offline Map OFF

Detailed Markers ON

Allowed MAC addresses

(a) *Options* view

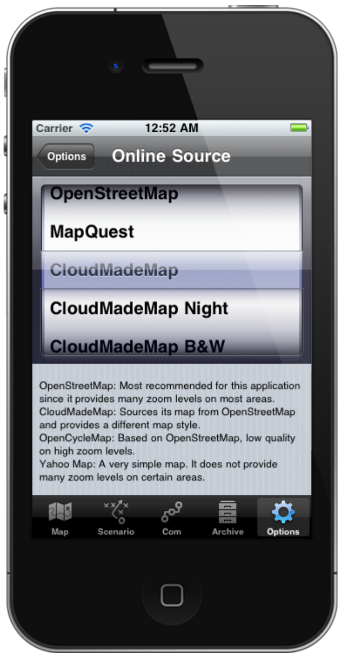


0012f0afdc39

0016ce364d46

Add MAC address:

(b) Managing allowed MAC addresses



OpenStreetMap
MapQuest
CloudMadeMap
CloudMadeMap Night
CloudMadeMap B&W

OpenStreetMap: Most recommended for this application since it provides many zoom levels on most areas.
CloudMadeMap: Sources its map from OpenStreetMap and provides a different map style.
OpenCycleMap: Based on OpenStreetMap, low quality on high zoom levels.
Yahoo Map: A very simple map. It does not provide many zoom levels on certain areas.

(c) Online source selector view

Figure 3.8: Different parts of the *options* view on iPhone

3.4 Communication

This section describes the communication flow between the managing device and UAVs for the *Airborne Relay* and *Airborne Multihop Relay* scenario using the *Autonomous Search Mode*. Figure 3.9 illustrates the *Airborne Relay* scenario and shows messages used by the *Remote Control App*. The application states depending on the communication stand on the left side and the required messages structures are provided by the *libuavext* library. The communication flow for *Airborne Relay* is as follows:

1. The *Remote Control App* starts in an unconfigured state indicating that the participants of the *UAVNet* are unknown. As soon as the UAV is operating, it broadcasts *ping* messages every few seconds using UDP.

2. In the next step, the managing application listens for *ping* messages. After receiving, the UAV is discovered and the information is saved to the primary database of the application.

3. When the *Airborne Relay* scenario is defined in the *Remote Control App*, the knowledge about the UAV allows the subscription to the UAV's notification service, accomplished through a *control* message using TCP. After receiving the acknowledgement, the device is subscribed and receives the *notification* messages from the UAV.

4. The next step is to transmit the *submitStartConfiguration* message containing the type of the scenario, the positioning and searching mode as well as the allowed MAC addresses for the deployment. If the scenario contained the *Manual Search Mode*, the application will automatically transmit the previously set location of the direction waypoint without user action after receiving the acknowledgement of the *submitStartConfiguration* message. For the transmission of the direction waypoint the *sendFlightDirection* message is used. Then, the *UAVNet* deployment procedure is initiated and performed by the UAV.

5. When the scenario should be aborted, the *Remote Control App* transmits the *sendAbort* message. The acknowledgement of this message will initiate an automatic unsubscription procedure for every subscribed UAV.

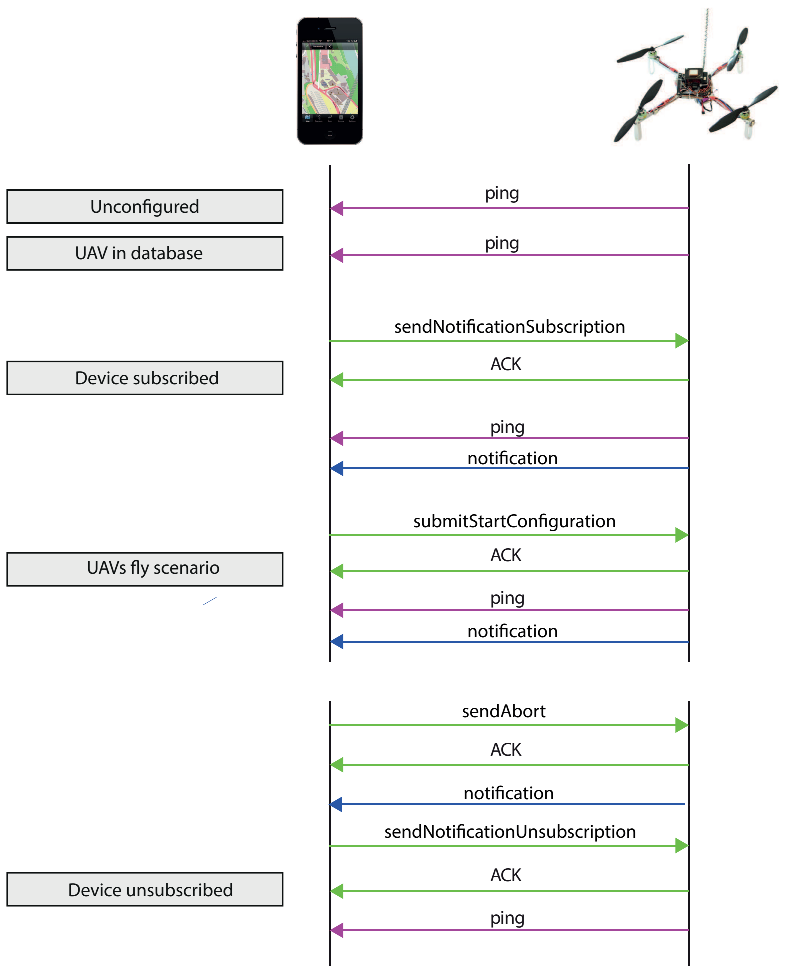


Figure 3.9: Communication flow between *Remote Control App* and a UAV with application states

The message flow for the *Airborne Multihop Relay* is illustrated in Figure 3.10. The difference to the *Airborne Relay* is that *ping* messages are received from multiple UAVs and multiple *sendNotificationSubscription* messages are sent as the *Remote Control App* must subscribe to every UAV separately. The *submitStartConfiguration* and the *sendAbort*, as well as the *sendFlightDirection* messages have to be submitted always to only one participating UAV. The UAVs inform each other through *ping* messages. How this message flow is handled with the GUI, is described in the following Section 3.5 and illustrated in Figure 3.12.

The *Monitor Mode* uses only *control* messages for subscription and unsubscription because an other device manages the *UAVNet* deployment.

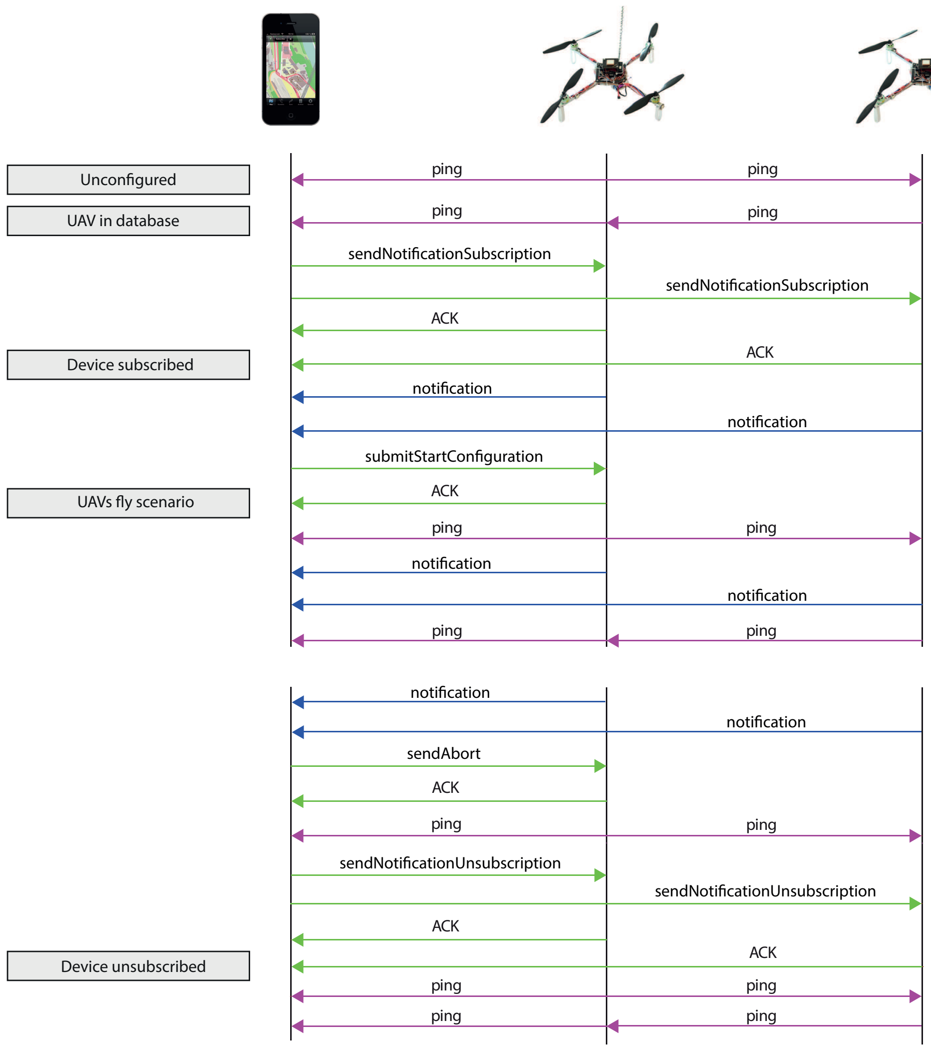


Figure 3.10: Communication flow between *Remote Control App* and multiple UAVs with application states

3.5 UAVNet Deployment with the Remote Control App

Only one instance of the *Remote Contol App* can manage a *UAVNet* scenario at a time. Other devices running the application within the same *UAVNet* can monitor the current deployment of the UAVs using the *Monitor Mode*. To set up a scenario, several steps must be carried out by the managing user. Figure 3.11 illustrates a typical setup for a *UAVNet* scenario. First, the UAVs have to be placed in the air because there is no automatic take-off procedure. The following list introduces the steps that have to be accomplished for a *Airborne Relay* scenario and Figure 3.12 shows how they are carried out with the GUI. These steps are similar for the *Airborne Multihop Relay* scenario.

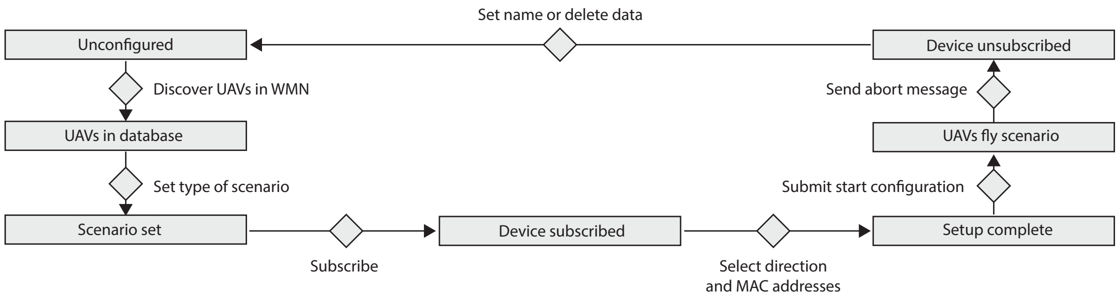


Figure 3.11: Performing a *UAVNet* scenario with application states

1. After establishing a connection to the WMN provided by the waiting UAV, the *Remote Control App* is started by the user. When the application's startup procedure is finished, it is in an unconfigured state as there is no *UAV*, *Station* or *Flight* object in the database marked as active.

2. Because the UAV has to be discovered prior to the transmission of the *sendNotification-Subscription* message, the user switches to the *communication* view. The connect button starts the listening thread for *ping* messages. After a few seconds the *Remote Control App* receives the first broadcasted *ping* message. Once the *Remote Control App* has received such a message, it creates a *UAV* object according the the provided information. The application will not create a *UAV* object if there exists already one with the same hostname. Instead, it marks the *UAV* object as active. Afterwards, the UAV is visible under the *UAVs* section of the *communication* view as shown in Figure 3.12(a).

3. The next step is to set up the desired flight scenario with the *scenario* view as illustrated in Figure 3.6. After the confirmation of the scenario type, a *Flight* object is created. It is also possible to set up the scenario before the detection of the UAVs.

4. Now the required information is available and the user can subscribe to the notification service of the UAV. On the *map* view the *state* button has to be pressed. This opens a view for the selection of the UAV as there might be multiple, previously discovered UAVs, but only one is needed. The view for the selection of the UAVs can be seen in Figure 3.12(b).

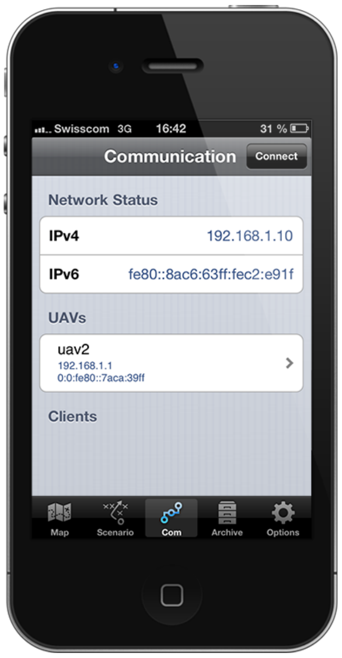
The checkmark next to the hostname indicates if the device is subscribed to the notification service. After subscription the user must confirm that he has finished the subscription procedure because it is still possible to unsubscribe and to subscribe to another UAV. For the *Airborne Multihop Relay* scenario, the selection view allows multiple simultaneously subscribed UAVs. Each received *notification* message gets processed by the observing thread into the database. The UAV will appear on the *map* view when the first of his *notification* messages is received.

5. When the subscription procedure is finished, the next state of the *Remote Control App* allowes to define the flight direction of the UAV depending on the search mode of the network scenario. If the *Manual Search Mode* was chosen, the *state* button enables the drawing mode and the user has to set a marker on the map to define the flight direction as illustrated in Figure 3.12(c). If the *Autonomous Search Mode* was chosen or the marker has been set, he can directly proceed to the selection of the UAV that receives the start configuration and the selection of the allowed MAC addresses.

6. Afterwards, the start configuration for the deployment of the *UAVNet* has to be submitted. The selection of the *state* button opens a view where the receiving UAV has to be chosen. The view for the selection of the UAV is shown in Figure 3.12(d). This is only intended for the *Airborne Multihop Relay* scenario as there is only one UAV in the *Airborne Relay* scenario, but the same view is used for both. The selection of a UAV slides in a table view where the user can select two MAC addresses similar to the selection for the subscription as illustrated in Figure 3.12(e). The confirmation of the selected MAC addresses triggers the transmission of the start configuration.

7. As soon as the UAV receives the start configuration, it starts the autonoumous deployment procedure according to the provided information. Figure 3.12(f) shows the map view for the *Airborne Relay* scenario with both clients and the UAV.

8. If the user wants to abort the deployment of the *UAVNet*, he submits the *sendAbort* message with the *state* button. Similar to the transmission of the start configuration, the user has to choose the UAV that should receive the *sendAbort* message for the *Airborne Multihop Relay* scenario. When the confirmation from the UAV has been received, the *Remote-Control App* automatically unsubscribes from all UAVs. Then the user is promted to set a name for the flight or he can instantly remove the data of the flight from the database.


Communication

Network Status

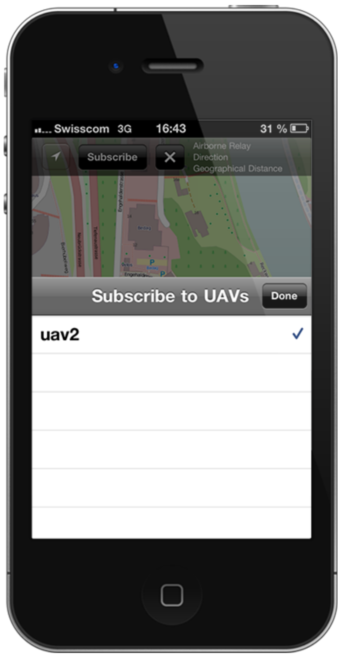
IPv4	192.168.1.10
IPv6	fe80::8ac6:63ff:fec2:e91f

UAVs

uav2
192.168.1.1
0:0:fe80::7aca:39ff

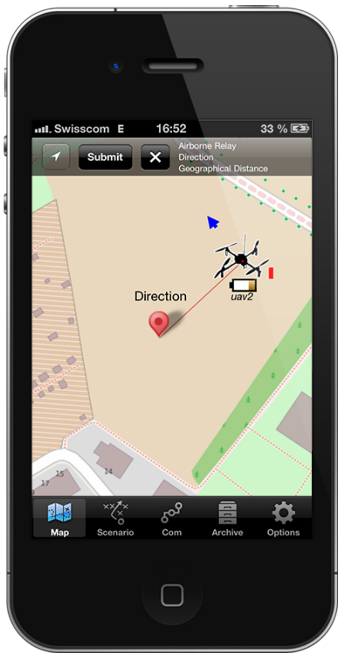
Clients

(a) Detection of UAVs

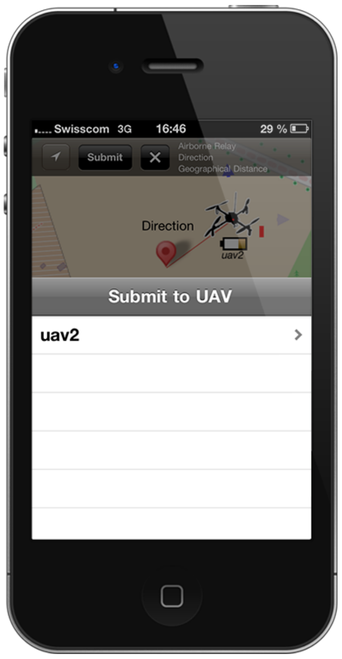

Subscribe to UAVs

uav2 ✓

(b) Subscription to a UAV

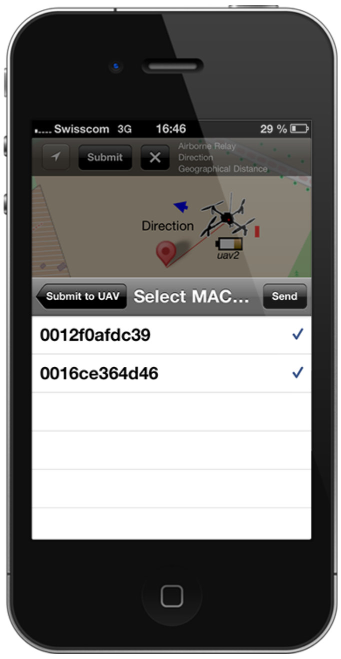


(c) Inserting of a direction marker


Submit to UAV

uav2

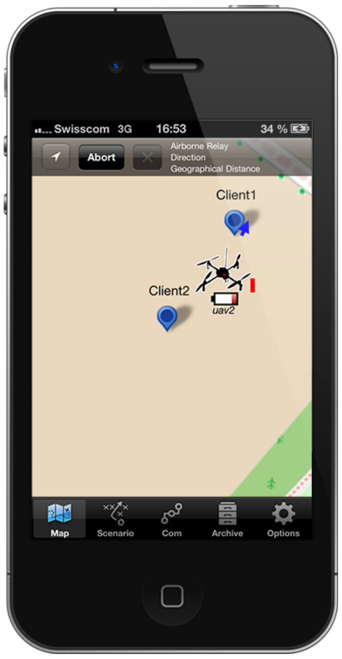
(d) Selection of a UAV to submit the start configuration


Select MAC...

0012f0afdc39 ✓

0016ce364d46 ✓

(e) Selection of MAC addresses



(f) *map* view of the deployment

Figure 3.12: Setup procedure for *Airborne Relay* on iPhone

Chapter 4

Conclusions and Future Work

4.1 Conclusions

This Bachelor thesis introduced the front-end application *Remote Control App* for *UAVNet*, a prototype for the deployment of an highly adaptive, mobile, airborne WMN using the IEEE 802.11s standard. This front-end application is capable to setup, to monitor and to manage the currently supported network scenarios of *UAVNet*. Since *UAVNet* is intended to be used in disaster scenarios for fast establishment of a communication solution, the use of a mobile platform, such as the iOS devices, supports this mobility concept. The design and layout of the *Remote Control App* as well as the graphical representation of the involved participants provide a user-friendly GUI for the overview about the status of the deployment and the management of the deployment procedures. The integration of locally saved map databases assures that a map can be shown even if there is no Internet access. The application states are simple for deployment of a scenario and can be accomplished by using only a small number of buttons on the interface. As there is always only one action that forwards to the next state, it can be assumed that even not very technically skilled people can understand and perform the deployment procedure with the application in a short time. These aspects provided by the *Remote Control App* support a future real-life usage of *UAVNet* in disaster scenarios.

Since the *Remote Control App* saves information of the received messages in its primary database, it provides exportable data that can be used for further analysis of the deployment process as well as for future developments of the *UAVNet*. The ability to support additional future extensions of *UAVNet*, such as the *Network Area Coverage* mode, are provided as well and could be achieved in the next steps. The use of the integrated libraries assures that the managing front-end and the *UAVNet* can evolve together.

Several flight tests during the development have shown that the deployment of the *UAVNet* can be successfully achieved with the *Remote Control App*, but there are still some shortcomings that have to be overcome. The take-off and landing procedures of the UAVs are not initiated by the *Remote Control App* and are currently unsupported by *UAVNet*. As a consequence, an additional remote control unit for the UAV-hardware is needed. This is a drawback for a real-life usage since the user has to use and to handle two different devices. Additionally, the whole deployment process of the *UAVNet* depends strongly on correct GPS values. When they are inaccurate, the *Remote Control App* does not show the real positions of the UAVs. Therefore, it can not be trusted that the saved location data is accurate to the really taken flight paths.

4.2 Future Work

The improvement of the *Remote Control App* strongly depends on the evolvement of *UAVNet*. There are many use cases that could be supported in future versions of the application, but some of them require extensions in the *uavcontroller* software, such as the *Network Area Coverage* mode. This mode requires a polygonal area that can be drawn by the *Remote Control App*, but additional adaptations have to be made once this scenario is supported by *UAVNet* since it is not defined how to transmit the nodes of the area. *UAVNet* requires additional collision detection and avoidance systems and a swarming behavior to achieve this type of scenario.

The MVC design ensures that the *Remote Control App* can be extended in an usable way and the application provides a starting point for future enhancements that could be an integrated cache configuration for the *Route-Me* library, an sophisticated download mechanism for map databases over the GUI and a geographical representation of other connected devices than the two clients of the *UAVNet* on the map. Since the *Remote Control App* is implemented in Objective-C, it would be possible to use parts of its code for a front-end application running on Mac OS X.

Additionally, an exchange of information over the WMN between a managing and multiple monitoring front-end applications, such as message exchange and Voice over IP (VoIP) calls, could be achieved. This would include a kind of detection mechanism for other connected devices running the front-end application similar to the *ping* message system of the *UAVNet*. In the current implementation, only one instance of *Remote Control App* can manage a scenario. It is possible to have multiple managing devices of a current *UAVNet* deployment, but this would require a security mechanism to either delegate the control to another device or to assure that only certain devices can manage a scenario. The first concept would require a detection and message exchange mechanism between the connected front-end applications. The second would require to log in to *UAVNet* as a managing device by a password or key. In the current implementation, the deployment of a network scenario is final because it has to be aborted to establish an other scenario, but the *UAVNet* and the *Remote Control App* could be adapted to change a scenario during a deployment, such as from *Airborne Multihop Relay* to *Network Area Coverage*.

Bibliography

[1] S. Morgenthaler, "UAVNet: A Prototype of a Highly Adaptive and Mobile Wireless Mesh Network Using Unmanned Aerial Vehicles (UAVs)," Master's thesis, University of Bern, 2012. [Online]. Available: http://cds.unibe.ch/research/pub_files/Mo12.pdf

[2] M. Honan. (2013) Apple unveils iPhone. PCWorld. [Online]. Available: http://www.macworld.com/article/1054769/iphone.html

[3] A. Brooks. (2010) iPhone OS Renamed iOS. World of Apple. [Online]. Available: http://www.worldofapple.com/archives/2010/06/07/iphone-os-renamed-ios

[4] E. Sadun, *Das grosse iPhone Entwicklerbuch*, 2nd ed. Pearson Education Deutschland GmbH, 2010.

[5] (2013) App Store Review Guidelines. [Online]. Available: https://developer.apple.com/appstore/guidelines.html

[6] J. Jun and M. L. Sichitiu, "The Nominal Capacity of Wireless Mesh Networks," *Wireless Communications, IEEE*, vol. 10, pp. 8–14, Oct 2003. [Online]. Available: http://networking.ncsu.edu/capacityWCM.pdf

[7] I. F. Akyildiz, X. Wang, and W. Wang, "Wireless Mesh Networks: A Survey," *Computer Networks Journal*, vol. 47, no. 4, pp. 445–487, March 2005. [Online]. Available: http://bcs.adi-share.com/contents/Wireless_Mesh_Networks_-_A_Survey428.pdf

[8] S. Misra, S. C. Misra, and I. Woungang, *Guide to Wireless Mesh Networks*, 1st ed. Springer Verlag London, 2010, pp. 325–326.

[9] I. F. Akyildiz and X. Wang, "A Survey on Wireless Mesh Networks," *IEEE Communications Magazine*, vol. 43, no. 9, pp. 23–30, September 2005.

[10] (2013) Unmanned Aerial Vehicles. [Online]. Available: http://www.thefreedictionary.com/unmanned+aerial+vehicle

[11] S. Morgenthaler, T. Braun, Z. Zhao, T. Staub, and M. Anwander, "UAVNet: A Mobile Wireless Mesh Network Using Unmanned Aerial Vehicles," in *Globecom Workshops (GC Wkshps), 2012 IEEE*. IEEE, Dec. 3–7, 2012, pp. 1603–1608. [Online]. Available: http://cds.unibe.ch/research/pub_files/MBZSA12.pdf

[12] J. Gentle and various developers. (2013) Route-Me: iOS map library. [Online]. Available: https://github.com/route-me

[13] G. Evenden. (2013) PROJ.4 - Cartographic Projections Library. [Online]. Available: http://trac.osgeo.org/proj/wiki/WikiStart

[14] (2013) Route-Me Map Framework. Universität Wien. [Online]. Available: http://athene.geo.univie.ac.at/project/route-me/html

[15] (2013) BSD License. Open Source Initiative. [Online]. Available: http://opensource.org/licenses/bsd-license.php

[16] (2013) "AOL - Products - Services - Local". [Online]. Available: http://corp.aol.com/products-services/local

[17] J. Gentle and various developers. (2012) Route-Me: Open source iPhone-native slippy map. [Online]. Available: http://code.google.com/p/route-me/w/list

[18] R. Krahl. (2013) Geo-OSM-Tiles-0.04. BerliOS. [Online]. Available: http://geo-osm-tiles.berlios.de

[19] F. Schröder. (2013) Map2Sqlite. [Online]. Available: https://github.com/magiconair/map2sqlite

[20] O. Ben-Kiki and C. Evans. (2013) YAML. [Online]. Available: http://www.yaml.org/spec

[21] (2013) OpenStreetMap - Wiki - Slippy Map Tilenames. [Online]. Available: http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames

[22] T. Staub, "Development, Testing, Deployment and Operation of Wireless Mesh Networks," Ph.D. dissertation, University of Bern, May 2012. [Online]. Available: http://www.iam.unibe.ch/~rvs/research/pub_files/St11.pdf

[23] iOS Developer Library. [Online]. Available: https://developer.apple.com/library/ios/documentation/general/conceptual/devpedia-cocoacore/MVC.html

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname:	Hänni Adrian
Matrikelnummer:	03-113-370
Studiengang:	Informatik
	Bachelor ☒ Master ☐ Dissertation ☐
Titel der Arbeit:	iPad/iPhone App as a Front-end for Prototype of a Highly Adaptive and Mobile Communication Network using Unmanned Aerial Vehicules (UAVs)
LeiterIn der Arbeit:	Professor Dr. Torsten Braun

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Bern 12.02.14

Ort/Datum

[illegible]

Unterschrift