LONE

December 2012 – version 4.1

ABSTRACT

Short summary of the contents in English...

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

GUI   Graphical User Interface

Frame Header Header containing metadata about a video frame such as resolution, and size

LIST OF CORRECTIONS

INTRODUCTION

When a crime occurs documentation is often needed in order to convict the person(s) responsible and to receive pay-out from an insurance company for the damages suffered. Documentation can be eye witnesses, forensic evidence, or video surveillance. Documentation in the form of eye witnesses or forensic evidence are unreliable sources, as it is not possible to guarantee it will be available. Video surveillance is however always obtainable in the case where one or more cameras surveil the crime scene, but it is not a perfect solution in its current form [32].

1.1 VIDEO SURVEILLANCE

Video surveillance is widely used to obtain documentation or evidence of a crime, or as a preventive tool. As an example there is one camera for every 32 people in the United Kingdom[25]. This number includes both cameras installed by the government to surveil London, but also cameras installed by businesses to secure their property. The high number of cameras is partially attributed to the high need for surveillance to prevent e.g. terrorist attacks in London, but additionally due to the technical difficulties of video surveillance.

A typical video surveillance solution consists of a set of cameras connected via cables to a "control station". Video cameras are mounted on a fixed location, and can therefore only surveil an area limited by their field of view, see Figure 1, and the environment in which it is placed.
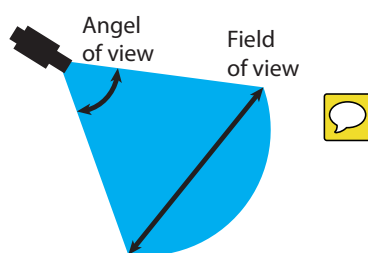
Figure 1: A cameras field of view

An environment can contain physical objects which limits the cameras field of view further, see Figure 2. In small environments a sufficient surveillance-degree can be achieved without too much difficulty

or cost [Appendix A]. Properly surveilling large areas is however problematic and can be expensive in regard to required hardware and installation costs such as digging down cables i.e. [Quote 1.1].

> " *We have issues surveilling large areas, such as mink farms, with traditional solutions such as surveillance cameras. The issues with a traditional solution for mink farms are that the establishment costs are very high due to resources used when digging.* "
>
> — Søren Ole Søndergaard, CEO of Lytzen IT

A

Video cameras have a limited field of view, both in terms of vision and range, and only within this field of view is the quality high enough be usable as documentation. This limitation can make it difficult and resource intensive to surveil large areas. The resource intensitivity stems from the amount of hardware required to surveil, referring to both cameras, and the cables needed to connect the surveillance system. The difficulty in surveilling the areas comes from the nature of an outdoor environment. In terms of surveillance an indoor environment is static, as there is a limited amount of entrances and the interior of the buildings in terms of walls and doors remain the same. In an outdoor environment the weather has to be taken into account. The weather can be foggy, rainy, and the sun can blind a camera if it is improperly placed. These conditions makes video surveillance of outdoor areas difficult. Both indoors and outdoors has some challenges that needs to be faced, such as physical objects being moved around creating blind spots etc.

1.2   CURRENT SOLUTIONS

This section will discuss some of the current solutions used for video surveillance, such as: regular camera, photo sensor, and dome camera. The strengths and weaknesses of each solution will be covered. The figures referenced in this section show the same area with a different surveillance setup to make them comparable.

In a regular camera solution, mounted cameras are used. These cameras surveil a limited area, and in order to cover a larger area, multiple cameras are needed. Multiple areas can be surveilled simultaneously with multiple cameras. A limitation with a regular camera setup is that the cameras are stationary. This means a cameras field of view can be limited by physical objects. As an example in Figure 2 it can be seen how an object placed in a cameras field of view can create a blind spot in an area normally covered by a camera.
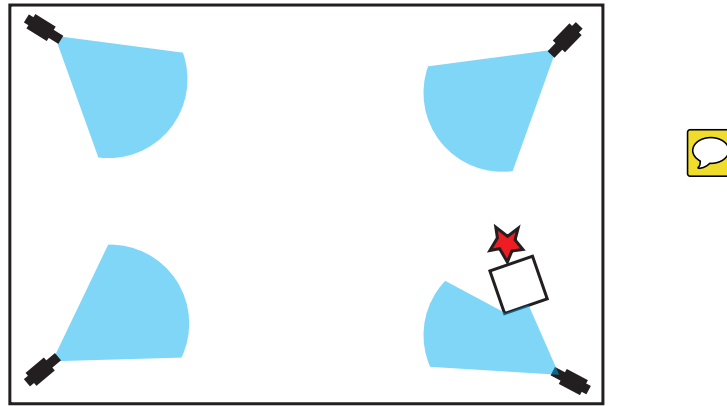
Figure 2: Example of a Regular Camera Setup

The red star in Figure 2 represents a critical object, that needs to be observed.

In a photo sensor solution the perimeter of the surveilled area is covered by both cameras and photo sensors. The purpose of the setup is to detect if anything enters the area, and then activate the cameras to capture it on video, as seen in Figure 3. A photo sensor solution can be used in combination with a regular camera solution, to surveil both the interior and the perimeter of an area. The advantage of a photo sensor setup is that the cameras surveilling the perimeter are only activated if the sensors are triggered, ensuring video footage is only recorded when it is needed. The disadvantage is that the cameras are still stationary, meaning a large amount of cameras are required to surveil a large area.

Figure 3: Example of a Photo Sensor Setup

The next solution centers around having a dome camera with a long field of view in the center. The perimeter is then divided into

zones, by e.g. light sensors. When a zone have detected an entry, the dome camera is rotated towards that zone to observe the area. Compared to the two previous solutions, the dome camera solution is limited to only observe a specific subarea at a time, as seen in Figure 4.
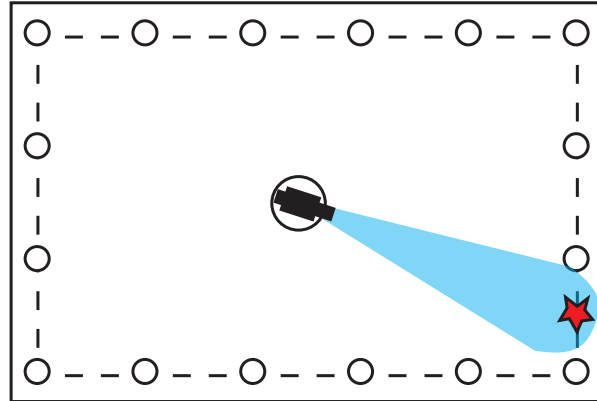
Figure 4: Example of a Dome Setup

1.3 DRONE TECHNOLOGY

As described in Section 1.1 the main problems with surveillance is the amount of hardware required, particularly cables, and cameras limited field of view. A possible solution is to switch from fixed cameras to movable cameras. A movable camera is one capable of moving freely in an environment under surveillance. This can be achieved with the use of drones with an on board camera. A drone in LONE is an unmanned aircraft also called UAV which is remote controlled. Drones capable of moving freely in an environment are capable of surveilling a larger area with fewer cameras than a traditional setup. Drones ability to move around objects also reduce the problems with blind spots and limited field of view. Surveilling an area with drones makes the surveillance responsive. Responsive surveillance means an area is only surveilled when it is needed. As an example an area might only be surveilled by a drone, when an alarm is triggered in the area i.e. the drone is not in the air at all times. The need for cables can also be removed by using drones, as they can be controlled using wireless technologies such as WLAN [13], radio waves or satellite [28]. In short a surveillance setup with drones requires less hardware than a setup with regular cameras, and surveillance can be made more responsive.

1.4 WEB FOR SURVEILLANCE

With the development and standardization of network technologies, video surveillance is becoming digitized. The digitization of video surveillance is important as it e.g. makes it faster to examine a video recording, and video cameras can be programmed to trigger alarms when they detect movement. Video recordings are examined faster as it is easy to skip through digital data compared to the analog data which would be stored on e.g. a VHS cassette. The programming of video cameras is possible due to the data being digital. It is possible to apply algorithms to examine the digital data, which could e.g. identify license plates on cars. The digitization allows for the use of the Internet. The Internet makes it possible to integrate surveillance solution with other computer solutions, that enables features such as backup or long distance observing. These features should be taken into account when designing a solution in this project.

1.5 PROBLEM STATEMENT

Video surveillance in its current form is stationary, meaning cameras cannot move. As a consequence of this, a large amount of hardware is required to properly surveil an area. This is problematic when a large area has to be surveilled, because the cost of hardware is high and the larger the area the more hardware is to be used. Drone technology offers a possible solution to this problem by making video surveillance dynamic through movable cameras. This possible solution yields the following preliminary problem statement:

How can drone technology be applied in a software application to improve the efficiency of video surveillance?

From the preliminary problem statement the following aspects must be considered:

- How to control a drone remotely over long distances
- How to provide video streaming of the drones camera through the web application
- How to make the web application scalable to support multiple drones and users
- How the make the system accessible from remote locations in a secure manner

For this project a AR Drone 2.0 Parrot, see Figure 5, has been acquired.
This yields the following problem statement:

Figure 5: AR Drone 2.0 Parrot. Picture from: http://guide-images.ifixit.net/igi/cI6EGVSLLjjDFATI.medium

How can drone technology be applied in a scalable web application to improve the efficiency and cost-efficiency of remote video surveillance of large outdoor areas?

1.6   CONTEXT

The project will be implemented as if it were a service offered by a company. Therefore the system must be developed as if it was to surveil several locations simultaneously. The hardware available limits the project is of limited quality and the number of drones is limited too. It is not possible to make a system usable in the real world using this hardware. This means the system will be developed to be a proof of concept and must be able to surveil several locations simultaneously.

ANALYSIS

In this chapter the problem domain for the problem statement defined in Section 1.5 is analysed. The problem domain is video surveillance of large outdoor areas. In the analysis the potential issues with implementing the posed solution are identified and presented. Based on the analysis a set of requirements to the application is presented and based on these a set of use cases defining the functionality of the system is constructed.

2.1 PROBLEM DOMAIN

The problem domain for the application is video surveillance of large outdoor areas. The solution investigated in this report is to use a drone with a camera instead of stationary cameras to handle this. The solution is intended to be used commercially by different businesses that may each have several different areas or locations that needs to be survelied.

Video surveillance using drones has a set of challenges which must be considered. The connection to the drone must be stable at all times to make it controllable. If the connection to the drone is lost the drone might crash, which must be avoided. The large outdoor areas makes it a requirement that the connection is stable at long range. Additionally the connection must support interaction with other services, making it possible to remotely control the drone. Furthermore the system must be scalable to include several locations, meaning several drones, and users simultaneously.

For video surveillance to be useful the video must be stored. It can be stored directly on the drone, or it can be stored externally. Storing it on the drone increases the hardware requirements for the drone, as it must be ensured the data is not lost should the drone crash. The alternative is to stream the video feed via the connection to the drone and store it externally. This reduces the requirements to the drone, but increases the requirements to the connection as the video feed must be usable.

There are two legal aspects to consider with regards to video surveillance. Firstly there are laws that limit the areas on which it is allowed to video surveil [20] in Denmark, where this project is developed. This is not an issue for video surveillance with stationary cameras, as

9

they are simply placed in locations that ensure they do not violate these laws. With movable cameras it is not trivial to ensure the video surveillance is done within the given laws. Furthermore the system must be secure. In this context secure means that access to the sensitive information is restricted, both internally and externally

The main problems present in the problem domain are therefore as follows:

- Wireless short-distance (with the drone, see Appendix B) and wired long-distance communication (back to the user) with a drone. This includes controlling the drone.

- Wireless short-distance and wired long-distance video streaming. The drone must be able to send back the video feed from its cameras. Therefore the system must also be able to transmit a video feed in real-time over long distances.

- Permissions- and access control in the web application. Includes both access to the system and access to specific drones.

- Scalability - the system must be scalable in terms of users and drones.

From these problems a set of requirements for LONE can be derived:

- A user must be able to login to the system.

- The system must have security measures in place to make sure any user only sees what he is permitted to.

- A drone must be able to send its video feed to the web service.

- A drone must be controllable from the web service.

- The system must support multiple drones and users.

These requirements are the base line for the use cases, which will be used to define the functionality of the system.

2.2   USE CASES

The functionality of the system will be defined through use cases, describing how we expect that a user of the system would desire to interact with the system. The use cases are derived from the requirements described in Section 2.1 and from the use case domain described in the following paragraphs. The use cases can be seen in

Table 6.

Along with the requirements mentioned in Section 2.1, a number of assumptions about the usage of LONE form the basis of the use cases in Table 6. These assumptions are our expectations to how the system will be used and therefore how to model it. We expect that a number of organization each will have a number of users connected to the system. At the same time we expect that there will be more users than drones present in the system, and that user-specific permissions to various elements of the system is needed, such as controlling a drone.

The terms *User*, *Drone* and *Permission* in context of LONE is therefore introduced. These regulate the relationship between one user in the system and a drone that he wishes to interact with via granted permissions.

An example can be an organization that has 10 employees to take care of the security and that this organization has 3 drones in LONE. All 10 employees must have access to all 3 drones with user-specific permissions.

We expect multiple users from the same organization to need the same permissions in LONE. In the case with 10 employees from the organization that needs the same permissions, it may make sense to collect the permissions for the 3 drones in a single Role and assign the 10 users to this role, rather than assigning all permissions for each drone go all 10 users. Therefore we introduce the terms *Company* to group users and *Rule* to group permissions.

These five terms – user, drone, permission, company & role – are the elements that forms the security- and permissions aspect of a users interaction with LONE and that are used in the use cases in Table 6.

ID	User Case
1	As a user I want to be able to login into the system.
2	As a user, when I am logged in, I want content shown based on my privileges.
3	As a user I want to pilot a specific drone.
4	As a user I want to view the video feed of a specific drone.
5	As a user I want to be able to grant or revoke other users the privileges that I am an admin of.
6	As a user with rights I want to change the settings for a specific drone.
7	As a user I want to be able to add and remove a drone to a company.
8	As a user I want an overview of all drones my privileges grant me access to.
9	As a user I want to be able to log out.
10	As a user with rights I want to be able to create and edit users.
11	As a user with rights I want to be able to create a new company.
12	As an owner of a company, I want to be able to edit the company.
13	As a user I want to be able to edit privileges, that I am allowed to edit, for another user.
14	As a user I want to be able to become a customer.
15	As a user I want to be able to add and remove privileges from a role.
16	As a user I want to be able to add roles to users within the company I am allowed to do so.
17	As a user I want to be sure that while piloting a drone nobody else can pilot the same drone.

Figure 6: Use Cases

The use cases define a set of objects which must be present in LONE in order for the use cases to be implementable. The objects reflect elements in the problem domain which must be modelled in the system. The objects are *users*, *drones*, *companies*, *roles*, and *privileges*. This leads to the dependency diagram in Figure 7. All objects are dependent on privileges, since they either provide or need privileges in order to be used. Everybody using LONE are classified as *users*, including customers, owners of companies, system administrators etc. However as reflected in the use cases *users* will not have unrestricted access to the system, as they depend on privileges. Access to the system will be restricted through *privileges*. As defined in use case #8 privileges grants access to functionality, meaning a *user* will not have access to anything by default. A *drone* refers to a physical drone. *Companies* is used as a grouping of associated *users*, *drones*, and *roles*. As an example a company might purchase a set of drones for surveilling their property. The company would need a set of users for its employees, and a set of privileges granting access to its drones. The *company* object would handle this grouping. Use case #15 defines a role as at set of privileges, that can be associated with a *company* and granted to *users*. Roles make the process of granting frequently granted privileges easier. This becomes easier as only one link have to be made to each user, instead of multiple links to multiple users.
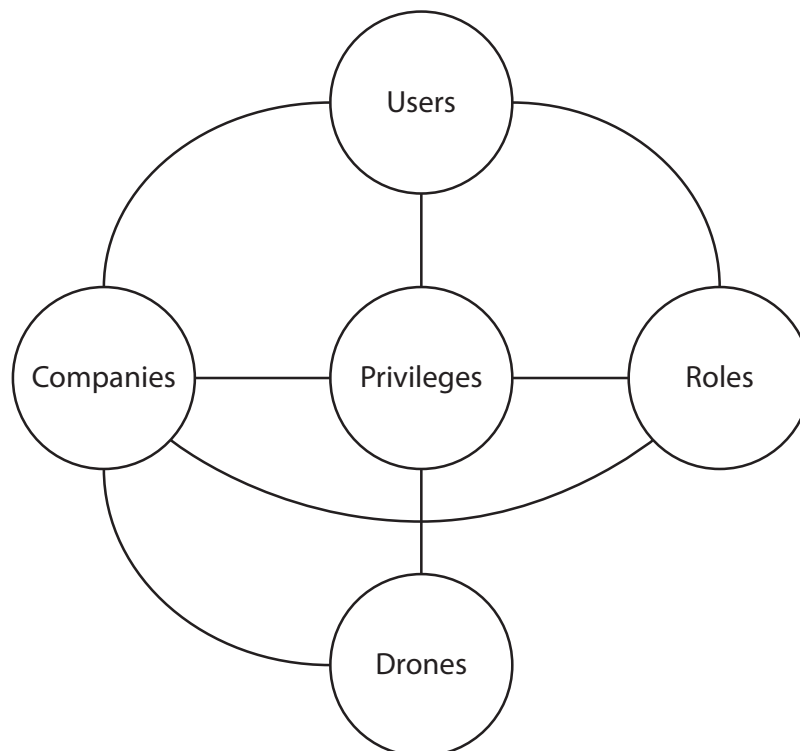
Figure 7: Element Diagram

==As== video surveillance is concerned with sensitive information, security in LONE is important. This is reflected in ==use case #1 and #2==. User authentication is used as the user must login to access to the content of system. Access is then further restricted with *privileges*, as the user's access to content is based on his privileges. Privileges must therefore be designed to be applicable to all ==aspect== of LONE, and be able to restrict access to all parts of the system. Restricting users access to certain parts of the system once they have gained entry is however not sufficient security for such a system. ==It must also be considered how the system can be made secure from external attacks==.

2.3 DEVELOPMENT METHOD

The choice of development method is based on previous experience and which method best suits the project. Which development method best suits a project is determined by the circumstances under which it is being done. The project is being done by a fixed size group of five members with a hard deadline. All participants in the project have similar experience with software development methods, both agile and traditional methods. The problem domain for the project is well defined, as there are no uncertainties in regards to the required functionality of the system. The functionality defined by use cases does however contain unknown such as video streaming, a distributed system, and communication with a drone. These unknown subjects creates an uncertainty about how to implement the desired functionality. Other factors to consider is the participants motivation, which can be affected by the chosen development method.

From the circumstances it can be derived that a traditional development does not fit the project. As the solution domain is unknown, creating a large upfront design is undesirable and might not be possible. Furthermore the hard deadline means that with a traditional method there is the risk that very little, or even none of the functionality is implemented, as the design phase might become to long or complex due to the uncertainties in the solution domain.

Therefore the best match for this project is an agile development method. The useful aspect of the agile development methods is their iterative nature. An iterative development method allows for refactoring, redesign, and most importantly for this project, the possibility of not implementing some functionality to meet the hard deadline. The agile development methods considered for this project are Extreme Programming *(XP)* and Scrum [24].

Both development methods have practices useful for the project. Therefore the development process for the project is a combination of the two. The practices taken from Scrum are:

- Sprint planning meetings

- A Sprint backlog of the tasks for the current spring

- Daily stand-up meetings

The practices taken from XP are:

- Pair-programming

- Acceptance tests

2.4 SYSTEM DEFINITION

LONE is a web service for surveilling large areas in a dynamic way. The system will be based on a drone which have the possibility of recording live video as previously stated in section 2.1. In this rapport a drone is defined as a device that is capable of flight, video recording, and video streaming. The service will be accessible via a web browser, where one or more users will be able to view the video feed from one or more drones and control their movements. The drone is controlled through keyboard gestures with a streamed video feed providing visual information about the drone's physical environment. LONE connects one or more users to a physical drone through their browser. Before a connection is established the user is authenticated in order to provide security features.

By accessing and authenticating on the website, the user should be able to view a number of drones that the user has access to. The user should be able to enter a detailed view of each of these in order to view the video stream and/or control the drone's flight remotely. The access control i.e. which users has access to which drones, is managed by an administrative user. Therefore the system will also implement different permissions to different users, user- and group structures.

2.5 ACCEPTANCE TESTS

The development method used in this project makes use of acceptance tests, see Section 2.3. Acceptance tests are associated with use cases, and must be passed in order for a use case to have been implemented. The acceptance tests are written based on the use cases and can be seen in Figure 8 and Figure 9. A use case may have several acceptance test associated with it. In Figure 8 and Figure 9 the column **Use Case ID** refers to the use case the acceptance is associated with. The use case IDs can be seen in Figure 6. The notion of acceptance tests will be of the format AT_X, where X is the ID of the acceptance test.

ID	Use Case ID	Acceptance test
1	1	The user is provided with a username and password form, that gives visual feedback based on the users action (logging in, failed attempt).
2	2	The user, after performing a valid login, is only shown content according to his privileges.
3	3	The user with rights is shown a page with an interface that allows him to pilot a specific drone.
4	4	The user with rights is shown a page with a window that enables him to see the video feed of a specific drone.
5	5	The user can successfully grant or revoke a privilege to another user.
6	6	The user is able to change the name of the drone.
7	7	The user is able to link a drone to a company.
8	7	The user is able to unlink a drone from a company.
9	8	The user is presented with a concise list of available drones.
10	9	The user is able to press a link to logout of the system.
11	10	The user with rights is able to create a new user via an interface.
12	10	The user with rights is able to edit an existing user via an interface.
13	10	The user with rights is able to deactivate an existing user via an interface.
14	10	The user with rights is able to activate an existing user via an interface.
15	11	The user is able to create a company via an interface.
16	11	The user with rights is able to remove a company.

Figure 8: Acceptance Tests 1

ID	Use Case ID	Acceptance test
17	12	The user is able to add users to the company.
18	12	The user is able to remove users from the company.
19	12	The user is able to add new roles to the company.
20	12	The user is able to edit existing roles in the company.
21	12	The user is able to remove existing roles from a company.
22	13	The user with rights is able to grant his own privileges to another user within the same company.
23	13	The user with rights is able to remove privileges from other users within the same company that he is able to grant them.
24	15	The user with rights can add privileges to the role.
25	15	The user with rights can remove privileges from the role.
26	16	The user with rights can add users to roles.
27	16	The user with rights can remove users from roles.
28	17	The user is not able to pilot a drone that is already being piloted.

Figure 9: Acceptance tests 2

DESIGN

This chapter will cover the design of the application by solving design problems, based upon the analysis, of the problem statement. The design problems are unfolded by the use cases and acceptance tests from the analysis process. The notion of acceptance tests will be of the format AT_X, where X is the ID of the acceptance test. The process of creating a solution includes discussion and reasoning behind the choices made throughout the design process.

3.1 APPLICATION STRUCTURE

Web applications have a single point of entry, a hostname, that is translated into an IP address. The IP address points to a server, which then handles HTTP requests (requests) from users, these users will be denoted as Browser or B. This means that a server solution is needed. Server solutions can either be singleton or distributed with multiple servers. Singleton server solutions are based on having one server that handles all incoming requests. This means that singleton server solutions are vulnerable to DoS, through HTTP requests, but provides consistency.
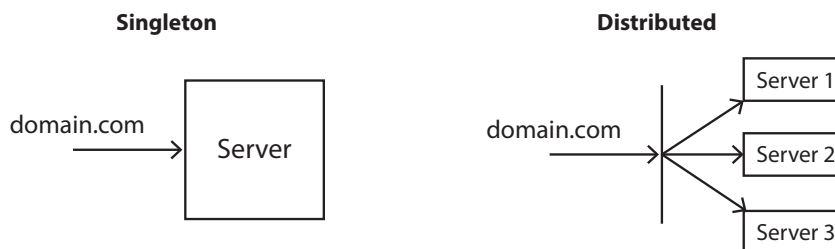
Figure 10: Illustration of a singleton and distributed server solution.

Distributed server solutions contain several servers possibly on several locations, meaning that DoS will be harder to perform, since there is no single point of failure. However, this solution cannot ensure consistency and adds complexity. The two solutions are displayed in Figure 10. A singleton solution has been chosen due to; consistency and simplicity. This singleton solution will be denoted as Master or M. Figure 10 only shows a single B, as it represents a single B's look at the system. It should be noted though, that M is capable of handling multiple B's at the same time.

The application is required to handle communication with drones, as defined by AT_1. In case of the communication being disabled, this might end up with a drone crash. The technical limitations of antennas for digital wireless communication, set a constraint for the availability of drones. Assuming there exists no antenna to cover the physical area of our problem domain, it is necessary to have a distributed antenna setup in order to cover the whole physical area of the problem domain. This leads to the structure shown in Figure 11.
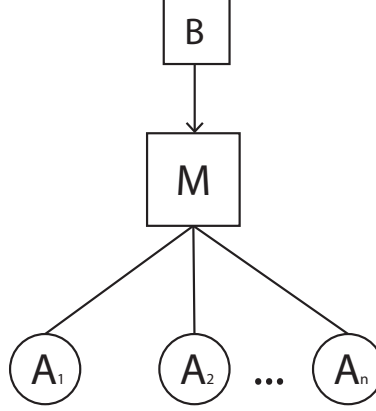
Figure 11: Antenna structure.

If all *B*'s communication with the drones goes through *M*, this would create a single point of failure. This means that, if *M* crashes at any point, all communication with the drones will be disabled. Providing the antennas with processing power and opportunity to communicate directly with *B* would solve this issue. If an antenna crashes only the drones connected to that antenna would be disconnected, leaving the drones connected to other antennas untouched. This could be achieved by distributing some of the communication from *M*. This is solved by combining the antennas with distributed processing units, which will be denoted as Slaves or *S*, as shown in Figure 12.

As *S* has a dynamic network position relative to *M*, this enforces that *B* is not able to communicate directly with *S* before getting network information about *S* from *M*. This communication is illustrated by a dashed line in Figure 12.

The acceptance tests, e.g. AT_{17}, AT_{24}, and AT_{25} enforce a constraint that requires *M* to be able to store data based of the interaction of the user. As requests can happen asynchronously, a relational database denoted *DB* is an obvious choice to store this data, as it is able to receive data at any time and let other processes handle it later, hence
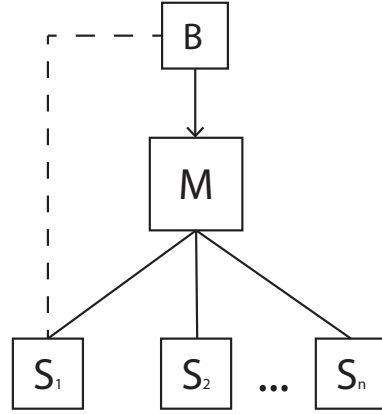
Figure 12: Slave structure.

helping to minimize potential data loss.

The response output of M needs to be dynamic based on the user, e.g. AT_2, which requires M to be capable of processing data and store it in the database. The processing unit that creates the dynamic response will be denoted W. The communication with the drones is handled by another processing unit, denoted D. If W were to handle requests from S and B, this would increase the load on W. By creating multiple request handlers, it is possible to have a process for each. W, DB, and S are processes, seen in Figure 13, which improves resource management. The resource management could be to constrain processing power for each process, or to distribute the processes on individual machines.
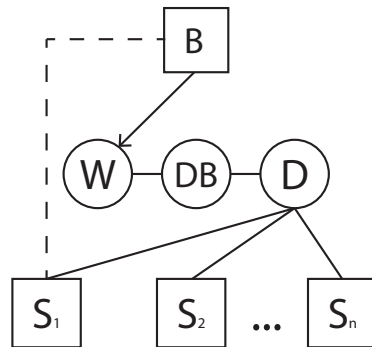
Figure 13: Daemon structure.

3.2   FUNCTIONALITY DISTRIBUTION

The functionality of the system can be derived from the acceptance tests. This section will cover what instances are responsible for each functionality.

The need for storing data is derived from AT_1, as previously described this is handled by the database DB, which is a part of M. AT_2 sets the need for a dynamic response to B, which requires processing of the stored data. This processing is handled by Web or W as the structure contains a singleton server solution.

Knowing that S has a dynamic location relative to M, this requires S to send a signal to the daemon or D in order for M to get the location of S on the network. D is then responsible for being able to receive incoming signals from S.

Displaying video is required of B by AT_3. The video displayed by B is a stream, which requires that S sends out a video stream. In order to control a drone, commands have to be provided to the drone, as required by AT_3. S is responsible for being able to receive commands and have the drone execute the given command. AT_{29} enforces security of the command handler. S is the only instance with a direct link to the drone, this makes S responsible for the command handler's security.

3.3   COMMUNICATION NETWORK

Having functionality distributed across several instances sets the need for communication between the instances. Based on Figure 12, which displays the dependency between B, M, and S this leads to the diagram shown in Figure 14.

The arrows represent a connection, each connection represent dataflow in the direction of the arrow. The type of data flowing in each connection and the reasons behind will be covered as each connection is discussed below.

Connection a covers requests made by B, which covers HTTP requests, which is a limit of web solutions that they must use the HTTP protocol. HTTP requests enable the user to view the web application through his browser, and to send information. In order to fulfill the request created by a, a response is needed which the connection b handles. This connection covers the flow of the dynamic content created by M, and static content such as: images, stylesheets, and multimedia objects. Connection b sends both the static and dynamic content back to B to be displayed for the user.
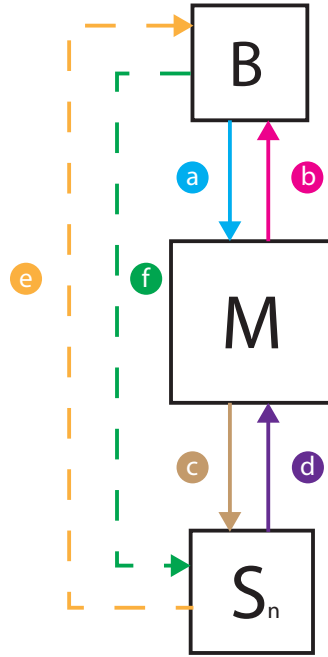
Figure 14: Dataflow diagram of LONE

The connection d handles the signal described in Section 3.2. This signal covers the initial communication between S and M. M verifies the identity of S by S sending its own unique identifier. This unique identifier is a string that is hard coded into S and unique for every S. It is generated before a new S is shipped and ensured that it does not match any existing identifiers. All unique identifiers of each S is known to M. This allows M to determine the identity of each incoming signal. When M receives a signal, and verify the identity of S, the source location of the signal is stored in M making M able to know the location of S.

Since requests from B can happen asynchronously and from dynamic locations, it can create a problem that the identity of the connection made from S to f is not known. AT_{28} sets the requirement of differentiating between incoming connections to S. If the communication described in connection e and connection f were running through M, this would be no problem as security would be handled by the already existing sessions on M. The connections e and f are required to transport a large amount of data. This data covers a video feed of the drone's video camera and commands for the drone. This would increase the load on M, which is not desired as M is assumed to have a high load already. However, since connection e and connection f are not running through M, this leaves the problem of which B, S should

listen to.

The solution chosen to solve this problem is to make *S* create a randomly generated key (session key), that contains 40 characters and is unique relative to *S* stored on *S* itself. When *S* receives incoming commands, *S* verifies whether or not the key received along with the command is equal to the one locally stored on *S*. If this is the case, *S* classifies the received command as valid and performs the action. The session key is delivered to *B* from *M*, as *M* is able to verify the identity of *B* through the session.

The connection *c* covers requesting a session key by *M*. As *M* has a static location, *S* is able to verify the identity of *M* based on its location. Connection *d* covers the response of the request received in *c*, and *M* is able to verify the identity of *S* based on its location.
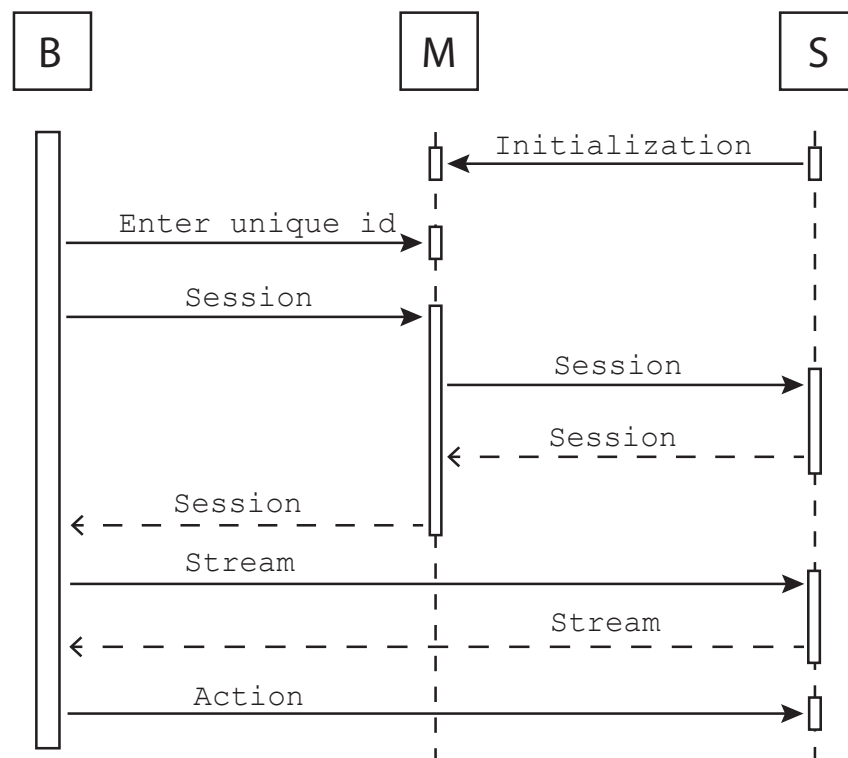
Figure 15: Sequence diagram of the communication network between *B*, *M*, and *S*

1

The functionalities of the described connections, can only happen in a sequence as illustrated in Figure 15. Each message in the sequence diagram can be done multiple times, however above messages are enforced to be performed atleast once in order for the given mes-

---

1 FiXme Fatal: Check UML notation.

sage to be performed.

3.4 PRIVILEGES

The need for restricting functionality is fulfilled by the privilege concept described in Section 2.2. This section will examine this functionality from an interaction perspective. Assuming there is a set of users U, and a set of privileges P to be assigned to each user in U. This leads to the following formula, where is I is the amount of interactions required by the user. The amount of interactions should be low therefore the lower the better.

$$I_1 = |U| * |P| \tag{1}$$

Assuming $|P| > 1$, $|U| > 1$, and both $|P|$ and $|U|$ will increase, then I_1 has a quadratic growth. However, there is another solution, where the privileges are grouped in a role, and the users are assigned to the role.This means that each privilege in P, have to be assigned to the role r and then r can be assigned to each user in U. The amount of interactions needed in order to create a role with the privileges P, is equal to I_1 where $|U| = 1$, this leads to:

$$I_r = |P| \tag{2}$$

Assuming that assigning a role r to a user requires 1 interaction. This leads to the following formula for calculating the amount of interactions using the role solution.

$$I_2 = |U| + I_r \tag{3}$$
$$I_2 = |U| + |P| \tag{4}$$

It is apparent that I_1 grows at a higher rate than I_2.

Based on the use cases, the assumption of roles in the problem domain having the exact same privileges for every user cannot be made. An example: The users A, B, C, and D share the same role as system administrator, but the users A, C, and D may view the log while user B may not. However, all users are categorized as the same, as their common role's privileges may change.

These exceptions must be handled by the system. A solution could be to put all users in a system administrator role, with all privileges that a system administrator should have, and then define that user B should not have the privilege to view the log, i.e. create an exception

for user B. Another solution would be to exclude B from the system administrator role, and grant him every privilege that he needs through user specific privileges. The issue with this approach is if the privileges for the system administrators change then these changes must be done for both B and the system administrator role. A third solution would be to have the system administrator role act as a lowest common denominator, and then add extra privileges separately, e.g. add the privilege to view the log to A, C, and D separately.

The system should be capable of handling all solutions, in order for the users to use the solution that fits their specific scenario.

In order to calculate the amount of interactions used for each solution a few functions will need to be introduced.

It is assumed there exists a $p(u)$, which returns the set of privileges that should be granted to a user u.

$$p(u) \subseteq P \tag{5}$$

It is also assumed there exists a $p_m(u)$ that returns the set of privileges a user u should not be granted from P.

$$p_m(u) = P \backslash p(u) \tag{6}$$

Furthermore the users that do not have all privileges within P will be denoted by u_m and users that have all privileges within P will be denoted by u_a.

Given the solution, where users are granted all privileges in a role and then revoked privileges through exceptions, the amount of interactions can be calculated with the following equation:

$$I_e = |U| + |P| + \sum_{i \in u_m} |p_m(i)| \tag{7}$$

The amount of interactions for the solution, where users that should not have all privileges of a role are not added to the role and instead given every privilege separately, can be calculated with the following:

$$I_{rm} = |U| - |u_m| + |P| + \sum_{i \in u_m} |P| - |p_m(i)| \tag{8}$$

Before describing the equation for the third solution it is necessary to introduce another function, $p_{lcd}(U)$ that returns the set of privileges that corresponds to the lowest common denominator within a set of users.

$$p_{lcd}(U) = p(u_1) \cap p(u_2) \cap \ldots p(u_n) \qquad (9)$$

With $p_{lcd}(U)$ defined it is possible to find the amount of interactions for the lowest common denominator solution. For this solution a role is created, which contains all common privileges, and the variable privileges are granted individually. It can be calculated as follows:

$$I_{lcd} = |p_{lcd}(U)| + |U| + \sum_{i \in U} |p(i)| \setminus |p_{lcd}(U)| \qquad (10)$$

3.5 OBJECT MODEL

This section will cover the design choices behind the application's object model, derived from the use cases in Section 2.2. The UML object model diagram can be seen in Figure 16, and is guided by the UML standard guide written in cooperation by several different software companies [18].

The UML attribute notation in this report is:

- PK attribute - means that the attribute is a primary key.

- FK attribute - means that the attribute is a foreign key.

- attribute : type - all attributes will have a type e.g. `id : int`.

The system will contain the following objects: *Affiliate Privilege*, *Company*, *Drone*, *Privilege*, *Role*, *Session*, and *User*. The model of the objects and their relationships can be seen in Figure 16. Each object covered is represented in the object model diagram with its name in lowercase and pluralized, e.g. *User* object → *users*. If an object's name consists of several words the spaces will be replaced by underscores, e.g. Session Key Task becomes session_key_tasks. The relationships between objects can either be implicit with a line directly from one object to another, or explicit with a simple or rich relationship model between the two objects. Simple relationship models are models without a PK. They are represented in the model diagram with both objects in lowercase and pluralized, e.g. roles_users. Rich relationship models have a PK. This kind of relationship model does not have a predetermined naming convention, therefore it can be anything as long as it does not collide with the other model names, e.g. user_privileges.

Access to and actions of the system is restricted. This restriction is based on the identity of user in the system. An instance of the *User* object represents a user to the system. The *User* object has the attributes
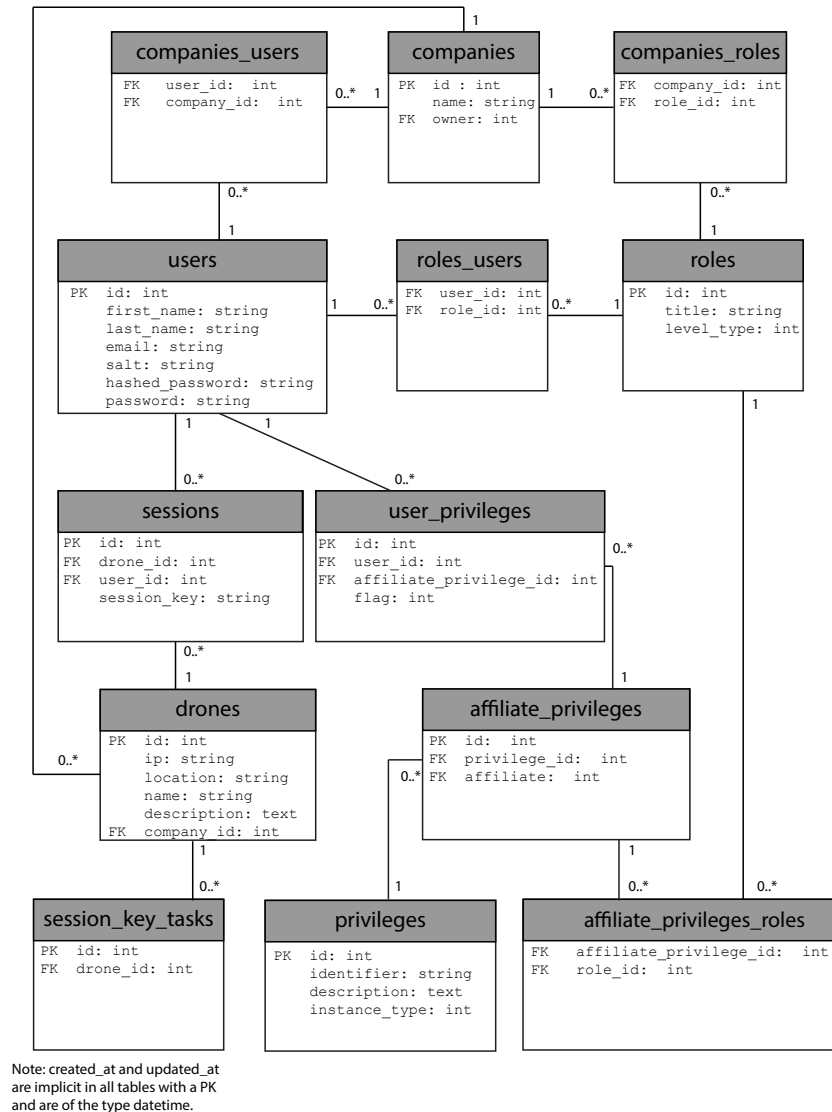
Figure 16: UML Class Diagram of LONE

email and password, as seen in Figure 43 in Appendix E.1, these combined form the login credentials needed for a user to authenticate his identity towards the system.

The email is publicly available, which leaves the password to be protected in order for user to protect his identity in the system. The password entered by the user is concatenated with a salt and hashed through a SHA-1 hashing algorithm to be stored in hashed_password. The hashing algorithm provides a way for storing the password without having its exact value, as this could leave to a security flaw. Having access to passwords of users directly, e.g. through a hacker attack, would allow the hacker to login into the users account, which is unintended for the system. When a salt is concatenated before hashing it makes it harder for the hacker to gain the original passwords, e.g.

through a rainbow table [**?**].

Having a distributed setup with multiple *S*, each representing a *Drone* object, means that the network locations of these need to be known in order to communicate. The network location is stored in the `ip` field of the *Drone* object, as shown in Figure 44 of Appendix E.1. The physical drone have a unique identifier known as `name`, which makes the user able to identify a given drone.

The users of the system will be part of different companies, therefore there is a need for grouping users by a company. The *Company* object have an `owner`, which is a reference to a *User* object. This is required for the system to know which user have access to all *Affiliate Privilege* objects of the company.

An *Affiliate Privilege* object references a *Privilege*, and only when combined they form a unique key. This key unlocks a certain functionality, meaning that if the user does not have a relation to the *Affiliate Privilege* object, which represents the key, then the user is not able to unlock that functionality. *Role* objects give the possibility of grouping *Privilege* objects together. Each *Company* have a related role which contains all privileges, of which the company have full control. Full control gives the right for passing on a privilege, or disabling if the privilege is already granted. *Affiliate Privilege* objects have a field `affiliate` that links the privilege to an object of the type declared by the referenced *Privilege* object's `instance_type` field.

The *Session* object represents an active pilot connection between a user and a drone. The `session_key` field provides a key that needs to be sent along with the commands. The *Session Key Task* object is an object used for *D* and *M* to communicate through *DB*, as shown in Figure 13 in Section 3.1.

In Section 3.4 solutions for handling privileges were discussed. From this discussion it was decided to model a system capable of handling all proposed solutions to fit scenarios of the users. The *User Privileges* relationship allows for connecting individual Privilege objects to a *User* object. This is could be either granting or revoking, i.e. exceptions, the privilege. The `flag` field represents a value of either 1 or -1; 1 is regarded as granted, and -1 is regarded as revoked.

3.6 MASTER

The structure of *M*, as shown in Figure 17 is derived from the functionality of Master described in Section 3.2. This figure shows three processes; Web, Database, and Daemon. The Database process, *DB*

can be any database engine capable of handling transactions, therefore this will not be elaborated further. The Web and Daemon processes will need further explanation, however.
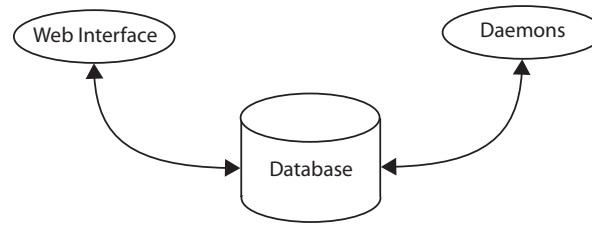
Figure 17: Internal structure of Master

The web process, *W* will be a web server using the Model-View-Controller (MVC) design pattern, see [27]. It consists of three layers, model, view, and controller. There are several frameworks in different languages that support MVC, which will be considered in Section 3.11.

The concept of MVC is decoupling the different layers to make it possible to develop them with some degree indepence. As an example a method for fetching data in the model, can be changed without affecting the view, as long as its output format remains unchanged. Decoupling the layers forces the developer to keep the code separated, keeping code meant for the controller out of the view. This separation means that the developer can isolate bugs to a specific layer, and can make debugging easier.

In Section 3.5 the object model is described. The models of *W*'s MVC are derived from this object model. Each model will have a direct mapping to a table in *DB*, and communication with the table in *DB* must go through the model. Some models needs extended functionality to implement the functionality defined in the use cases. This functionality involves e.g. authenticating users and privileges. The functionality of the models can be seen in Appendix C.

In order for a model to have a set of associated views it must have a controller. In LONE this includes models such as User, Drone, and Company, described in Section 3.5. There will furthermore be controllers with no corresponding model, e.g. an access controller, which will handle login and logout. Each controller in LONE with a corresponding model, i.e. Affiliate Privilege, Company, Drone, Privilege, Role, and User, will have Create, Read, Update, and Delete (CRUD) [6] as default actions. All the controllers' actions are listed in Appendix D

The daemon process, *D*, has two tasks and will be a process running in the background of *M*. One will handle the session communication with *S*, as described in Section 3.3. The other handles receiving initialization messages from *S*. This communications between *M* and *S* will be done using JSON (*JavaScript Object Notation*).

Session communication refers to the session key required to communicate with a drone. *D* will be written in the same language as *W*. *D* will handle session communication with *S* by continuously checking *DB* for new session requests. Session request are entered in a table in *DB* by *M* when a request for one is received from *B*, as illustrated in Figure 18. The session communication is handled through the *DB*, to account for the asynchronous behaviour of the web[30]. This makes it possible to run multiple instances of *D*, as they can all read the same table in *DB*. If a session request is present in *DB*, then *D* will request a session from the associated *S*. If *D* receives a valid session key from *S*, it saves this in *DB* and *W* sends it to the user. *D* should handle session requests chronologically, thus if several requests are present in *DB* then *D* will not perform all requests simultaneously. The second purpose of *D*, is to handle initialization messages from *S*. Each *S* is associated with a *drone* in the object model described in section 3.5. When *D* receive an initialization message from *S* it will create a *drone* object and store it in the *DB*. In the *drone* object the `ip`,`location`, and `name` will be retrieved from the initialization message send from *S*. The `location` is retrived from the IP and the `name` is the drones' serial code. If *M* receives an initialization message from a *S* with a changed IP, it will update the `ip` and `location` of the *S* in *DB*. The drones' serial code will be described in Section 3.7.

All communication between *M* and *S* is done using JavaScript Object Notation (*JSON*)[9]. The choice of JSON is described in Secion 3.11.

D will handle the session requests as a First In, First Out (FIFO) queue, to reduce the average time it takes to receive a session key after requestion one. However, since the response from *S* can be delayed for an arbitrary amount of time, there is no guarantee that the first session request to be handled is also the first session to be entered in *DB*. *S* will only return a valid session key, if it currently has no active sessions. Therefore at most one person will have a valid session key for communicating with *S* at any given time. How this session key communication is handled can be seen in Figure 18.

3.7 SLAVE

The functionality of Slave is as follows, derived from Section 3.3 and Section 3.2.

1. Establish and maintain a wireless connection to the drone.
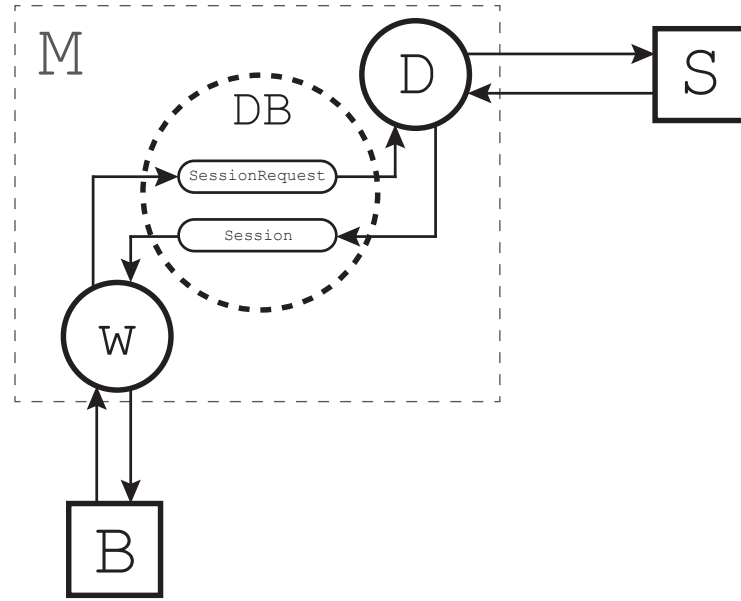
Figure 18: Session key communication.

2. Send initialization message to M when powered on, as seen in Figure 15.

3. Create and send session keys upon request.

4. Receive flight-operations from *B* and send them to the drone.

5. Read video feed from the drone and forward it to *B*.

The slave is connected to the drone and the Internet. Therefore two network interfaces are required, one of which must be a wireless network interface (WIFI-interface), as all the communication with the drone is wireless as described in Appendix B. *S* will automatically attempt to establish a connection to the drones wireless network when powered on. Following this the initialization message is send to *M*. When the initialization is succesfully completed, *S* is ready to use. Each *S* will have a unique identifier called serial code.

The initialization message is send from *S* to *M* as JSON, see Section 3.11. It will contain *S*' serial code. The initialization message is send to *M* when *S* is powered on.

A session key is generated on *S* when a request is received, see item 3. It is not be possible to communicate with *S* without a valid session key. The session key is generated on *S* to ease the load on *M*. As mentioned in Section 3.3 there can only be one active session key active on *S*. A session key is a 40 character string consisting of letters and numbers. This was chosen based on [22] which states that length is stronger than complexity when creating passwords. A session key of length 40 was deemed sufficient. Following the generation of a

session key it is stored on *S* and send to *M*. While a session key is active on *S* only messages containing it are forwarded to *S'* associated drone. The session key is send to *M* as JSON, see Section 3.11. Session keys will be deleted from *S* if the connection times out.

S receives flight-operations from *B* which must be forwarded to the drone, see item 4. The flight-operations must be read and converted to an `AT command` for it to be interpretable by the drone, see Appendix B.

Item 5 will be explained in Section 3.11, as already existing technologies will be used.

3.8 BROWSER

Users interact with the system by connecting to *M* through *B*. This connection is through an Internet browser as described in Section 3.1. The user will interact with the system through a web interface, which is retrieved from *M*. The functionality of *B* is described in Section 3.2. If *B* interacts with a drone, *B* must be able to display the drone's video feed, and allow the user to send commands to the drone. The communication with the drone is through *S*, therefore *B* must be able to communicate directly with *S*.

For *B* to interact with the system requires authentication to establish a connection with *M* or *W*. A session that holds the identity of *B* is stored on *M*, and it is used to uphold the users' authentication.

To ease the load on *M*, some of the computation is moved from *M* to *B*. *B* has two types of interaction with *S* as described in Section 3.3. To display the drones video feed, a video player must be present in *B*. To control the drone *B* must request a session key from *M* as illustrated in Figure 15. When controlling the drone the video feed must be available in *B*.

3.9 USER INTERACTION

The user interacts with LONE through *B*. AT_1 and AT_2 requires that the user must be granted access to the system before being able to interact with it. This access is granted through a valid login and privileges. When *B* initiates a connection to *M*, it returns a login view. This is done by the access controller, which is described in Figure 34 in Appendix D. The user enters his login credentials in the login view and *B* sends a login request to *M*. Following a successful login the user is redirected to the drone view, where the user is shown a menu and the available drones.

B can then interact with any of the models described in Section 3.5 that have associated views. The general interaction with these models is similar, and the interaction with drones is described as an example. When *B* requests to interact with drones, the drone controller returns

a view containing a list of all drones available to the user. This list contains all drones the user has access to based on his privileges. The users privileges are retrieved through the users model. For each drone in the system it is checked if the user has privileges that grants him access to the drone. From the drones list the user can edit a drone, or delete a drone. Drones are added automatically to the system through the initialization messages send by the *Slaves*. To make a drone usable in LONE, it must be added to a company as reflected in the object model, see Section 3.5.
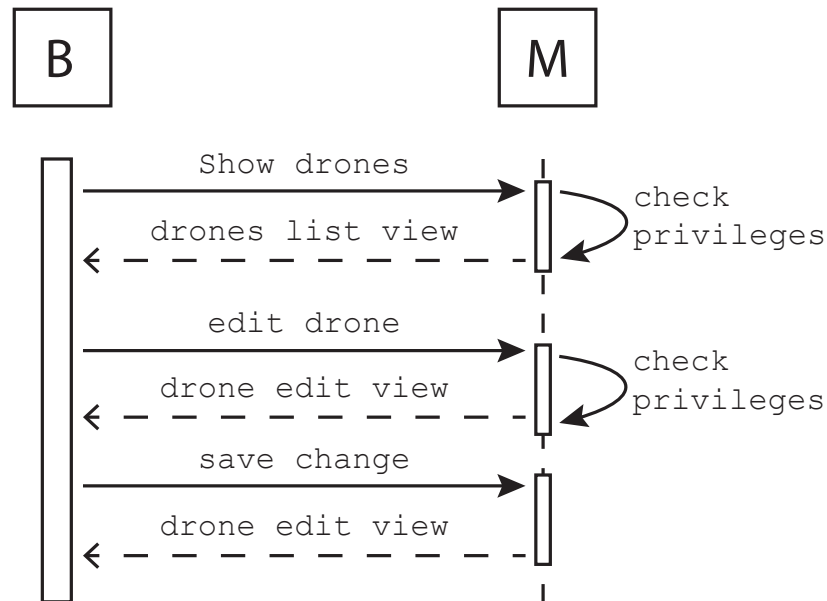
Figure 19: Illustration interaction between *B* and *M*.

The process of editing a drone is illustrated in Figure 19. *B* request the list of drones from *M*, and is returned a list of drones based on his privileges. *B* then requests to edit a drone and *M* ensures the user has needed privilege to edit the drone. If this is the case, *B* receives a view from *M* where the user can input data to update the drone with. The user can then choose save the entered data and *B* then sends it to *M* which saves them in the *DB*. *M* then returns *B* an updated edit view the drone.

As mentioned a user deletes a drone through the drones list sent to *B* by *M*. When a delete request is sent from *B* to *M* for a drone, *M* deletes all links the drone has with other object in *DB*. The drone is not deleted from the system, as described in Section 3.5

Some of the views require asynchronous behaviour, that allows the views to be updated without recomputing it on *M*. In this way some of the computation is moved from *M* to *B*, as described in Section 3.8.

From the drones' list *B* can request to pilot or view of the drones' listed. When *B* requests to pilot a drone the process of retrieving a

session key from the drones' associated *S* is initiated by the drones controller. *B* is redirected to a page containing the video player described in Section 3.8 when the session key is recieved. From this page *B* can view the drones video feed and send control commands to the drone as described in Sections 3.8 and 3.7.

3.10   USER INTERFACE

This section covers the principles and guideline used in order to create the Graphical User Interface (GUI). The layout for all views, that require authorization to use, was created with the use of the Gestalt principles [31]. This layout is a graphical guideline, as it contains graphical elements for the GUI, while having defined areas for dynamic content. The reason behind using a layout is to gain similarity, as described in the Gestalt principles. This is done to improve the user's ability to differentiate between when he or she is authorized and not authorized.

The layout used can be seen in Figure 20. *Content* of the layout is a dynamic placeholder for individual content of each view, while the menu remains the same.
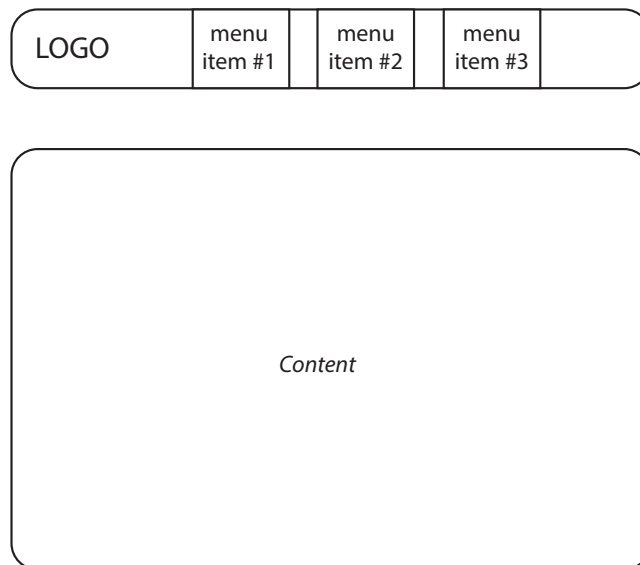
Figure 20: GUI layout.

Since the HTTP protocol is stateless and requires all parameters to be received through requests [12], it creates a behavior which might be counterintuitive to the user, e.g. when using the browser's back and forward functionality. Moving backwards in a browsing history which involved sending parameters causes the browser to resend these parameters. This might not be the intention of the user. For

instance, assuming a user wants to create two new roles. The user is presented with a list of the current roles in the company, and navigates to a view allowing him to create a new role, r_A. When the information about r_A has been filled in, the user submits this data by creating a HTTP request to W with parameters containing the information about r_A. W processes the parameters and creates the role r_A. Then W presents the user with a view showing r_A. Assuming the user now wants to navigate back to the view of all roles via the back functionality in the browser. The browser will perform his previous request, i.e. the request to create the role r_A, which W interprets as creating a new role r_B with the same parameters as r_A. One way to solve this problem is to create requests in the background of the browsers history through the asynchronous requests described in Section 3.8.

3.11   TECHNOLOGIES

The application structure described in Section 3.1 specifies LONE will consist of a web application on M and server-side scripting on M and the *Slaves*. In this section the development setup will be chosen, including both server operating systems and programming language. The design furthermore specifies some functionality which must be implemented using specific technologies.

3.11.1   *Server Operating System*

In this project two operating system families will be considered for use. Each family consist of a set of operating systems.

- Windows Server family
    - Windows Server 2003
    - Windows Server 2008
    - Windows Server 2012

- UNIX family
    - Linux
        * Debian/Ubuntu
        * Fedora
        * CentOS/RedHat
        * Various other distributions of Linux
    - BSD
    - Mac OSX Server

LONE will use Linux Debian as server operating system:

- Installation of new programs is quick through the Debian package manager.

- Linux Debian is open-source[10].

- Linux Debian is one of the most well-liked operating systems amongst system administrators, credited for its maintainability, size, ease to customize and community support [29].

The operating system chosen for the server-side applications in LONE is Linux Debian.

3.11.2 *Programming language*

Using Linux Debian as operating system on *M* and *S*, limits the possible options for programming languages.

The programming language must have the following:

- Support for network communication, see Section 3.3.

- Support for communicating with databases, see Section 3.1.

- Support for object oriented programming, see Section 3.5.

- Must be maintained to ensure the language does not become discontinued.

The considered programming languages were:

- Python

- Ruby

- Perl

- PHP

The language chosen for LONE is Ruby. Ruby was chosen for the following reasons:

- Ruby can be used as a scripting language.

- Ruby on Rails *(Rails)* provide a framework for web-development with Ruby.

Ruby is a object oriented scripting language.

Rails is an open-source web framework for Ruby[17], the following paragraph uses it as source. Rails provide a set of built-in features, such as native support for the MVC model and Active Records. Rails also contains features, which simplifies distributed development. An example is Migrations. Migrations are set of ruby classes designed to

make it easy to setup and modify databases. With migrations each developer can use a local database for development, instead of a shared database. If a developer makes changes to the database, he makes a new migration, shares it with the other developers, and by running the migrations their local database is updated to the latest version. Additionally migrations makes deployment of systems easier, as the database is automatically setup by running all migrations.

3.11.3 *Browser technologies*

A number of technologies are required in the browser to display the web application. As described in Section 3.8, *B* must be able to display a video feed and handle some of the computation to reduce the load on *M*. Reducing the load on *M* can be achieved using Javascript [7].

Another tool that helps reduce computation on *M* is: Asynchronous Javascript and XML *(AJAX)*. AJAX allow *B* to request a specific view asynchronously from the rest of the view, and using Javascript computed in *B*, handle the output from *M* and attach the information to the view. This also provides the asynchronous processing described in Section 3.9. This approach has several advantages:

- The content of a view can be updated without recomputing it. This is normally not possible with web-development, as the web is stateless[30].

- Using AJAX can improve usability of a web application.[5]

These tools ensure that *B* of LONE is be platform independent while providing a better user-experience for the users [5].

As described in Section 3.8 a technology must be used in *B* to display the drones video feed. The technologies considered for this are:

- Flash

- Silverlight

- HTML5

Flash was chosen. HTML5 was discarded as it is still a new technology, and therefore not fully supported by all browsers[11]. Silverlight applications are developed using .NET. Since the other parts of LONE are developed to work with the chosen Linux Debian server, Silverlight was discarded.

Flash is platform-independent and can be run as a plugin in the clients Internet-browser. The flash-application will be able to connect the user directly to the slave associated with a drone, enabling the user to view its video feed and control it.

3.11.4 *Streaming Technologies*

Developing a video streaming tool is without the scope of LONE as described in Section 3.7. Video streaming is therefore done using existing streaming technologies.

The video feed send by the drone to *S* is forwarded to *B* as seen in Figure 15. The video feed is displayed in a Flash application, as described in Section 3.11.3. Flash only natively supports video streams send via Real Time Messaging Protocol*(RTMP)*[4]. The drones video feed is H.264 encoded and send over TCP, as described in Appendix B Flash is compatible with H.264 encoded video [3]. The drones video feed is however encoded with the PaVE headers as Header containing metadata about a video frame such as resolution, and size (Frame Header), see Appendix B. Flash in incapable of reading PaVE headers. Therefore they must be removed before Flash is capable of displaying the video feed. To achieve this each *S* must contain functionality capable of reading and encoding the drones stream without PaVE headers and forward it over RTMP to the Flash application.

There are two approaches to achieve the streaming functionality required in LONE. One is using a single streaming technology with functionality to read the drones video feed and broadcast it over RTMP without the PaVE headers. The other is using two distinct technologies, with one reading the drones video feed and forwarding it to a streaming server. The streaming server reads the incomming video feed and broadcasts it over RTMP to *B* Both approaches depend on a technology with functionality to decode the PaVE headers. The technologies considered are the multimedia frameworks FFmpeg and Gstreamer, and the multimedia server C++ RTMP Server*(CRTMP)*.

"FFmpeg is a complete, cross-platform solution to record, convert and stream audio and video" [1]. It contains a multimedia streaming server called FFserver. FFserver is used for live broadcasts and is capable of decoding and enconding video and audio. It does not contain functionlity to decode the PaVE headers, but can output over RTMP. Therefore FFmpeg cannot be used to read the drones video feed, but can send the RTMP stream if given a proper input.

Gstreamer is a multimedia framework that uses a pipeline architecture. A Gstreamer pipeline is a set of plugins the video feed is processed by. As an example a video feed might be decoded and then encoded in a new format. Gstreamer does not have a RTMP server. It is however capable of sending a video feed to a RTMP server. Gstreamer does not contain functionality to parse PaVE headers. There does however exist an externally developed plugin for Gstreamer [21] which contains the functionality to parse PaVE head-

ers. The plugin is named Gstreamer PaVE parser *(paveparse)*.

CRTMP is a streaming server capable of streaming to and from a Flash application. It can receive a local stream send over RTP and broadcast it globally as RTMP.

Gstreamer is the only tool with functionality to parse PaVE headers, but as it does not contain a RTMP server it must be used in correlation with either FFmpeg or CRTMP. FFserver is not capable of reading a video stream outputted by Gstreamer [2], therefore CRTMP is used as the RTMP server in LONE

4

IMPLEMENTATION

In this chapter the implementation of LONEs' functionality is explained. This chapter does not cover the implementation of the entire project, but merely selected parts. It can be expected, that the implementation of all of the functionality that is not described in this chapter, is implemented as it is designed in Chapter 3.

4.1 STREAMING

In this section the setup of the two streaming tools GStreamer and CRTMP for LONE will be explained. Following the setup of GStreamer and CRTMP, the next step is reading the output stream and displaying it in a Flash application. Due to the PaVE headers, however, this is not possible.

4.1.1 *GStreamer*

As described in Section 3.11.4 the tool GStreamer is used for reading the drones' video feed and forwarding it to CRTMP. GStreamer is a pipeline based tool, and has a command line version which can be executed using `gst-launch-0.10`. The plugins the pipeline consist of are separated by a `!`. The first plugin in the pipeline is called a source element, and the final element a sink. The plugins between the source and sink element consist of a set of video and audio processing-, and data management plugins. A sample pipeline is illustrated in Figure 21.
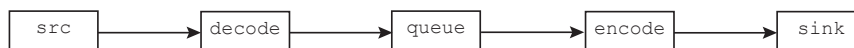
Figure 21: Illustration of a sample pipeline.

The source element is capable of receiving a video feed over a specific protocol from a defined destination. As described in Appendix B the drone outputs its video feed using the TCP protocol on port 5555, and its IP-address is 192.168.1.1. To read the stream, the GStreamer plugin `tcpclientsrc` is used. The parameters `host` and `port` can be set for `tcpclientsrc`. The first part of the pipeline can be seen in Listing 1.

The data available to the next plugin are H.264 encoded video with PaVE headers. The PaVE headers are parsed using the paveparse plugin [21], as described in Section 3.11.4. Paveparse removes the PaVE

41

4

[December 17, 2012 at 15:27 – classicthesis version 4.1]

```
1  gst-launch-0.10 tcpclientsrc host=192.168.1.1 port=5555
```

Listing 1: Partial GStreamer pipeline illustrating tcpclientsrc

headers and send the remaining H.264 data down the pipeline. Furthermore is ignores lost frames and discards frames sent out of order to reduce delay on the stream. This is necessary as the drone streams over TCP, and as described in Section [1] delay is not acceptable on the stream.

As specified in Section 3.11 the video output of GStreamer should be in FLV so that the Flash application in *B* can display it. Two pipelines that output FLV can be created. The first is using a H.264 decoder to create raw video data, and then encode it as FLV. The second is using FLVmux. FLVmux muxes audio and video streams into a single flv file [19]. However, as the H.264 data has no headers, due to *paveparse*, the H.264 decoder cannot decode the video. Therefore an FLVmuxer is used to create FLV output.

With FLV video outputted from the FLVmuxer, the sink-element can be added to the pipeline. As the video feed is forwarded to CRTMP, the sink element is rtmpsink. It has one parameter called location, which is the URL-address of the RTMP server with an extension specifying the source-URL of the stream on the RTMP server. The rtmpsink can be seen in Listing 2:

```
1  rtmpsink location='rtmp://0.0.0.0/live/myStream'
```

Listing 2: Partial GStreamer pipeline illustrating rtmpsink

The pipeline that forwards the video feed from the drones can be seen in Listing 3.

```
1  tcpclientsrc host=192.168.1.1 port=5555 ! paveparse ! flvmux ! rtmpsink
       location='rtmp://0.0.0.0/live/myStream'
```

Listing 3: GStreamer Pipeline with tcpclientsrc and rtmpsink

To handle plugins working at different speeds data management plugins are added to the pipeline. The plugin used is queue. queue is a data queue that queues data until e.g. the queue reaches a specified size [15]. The queue element is added between every plugin except

---

1 FiXme Fatal: Ref til analysis om hvorfor no delay på streamen er vigtigt, og check at sætningen stadig giver mening

`tcpclientsrc` and `paveparse`.

The complete pipeline can be seen in Listing 4:

```
tcpclientsrc host=192.168.1.1 port=5555 ! paveparse ! queue ! flvmux !
    queue ! rtmpsink location='rtmp://0.0.0.0/live/myStream'
```

Listing 4: Complete GStreamer Pipeline

4.1.2 *C++ RTMP Server*

As described in 3.11.4 C++ RTMP Server is used as the server tool between GStreamer and the Flash application in the *B*. CRTMP can be executed as a daemon or a console application. It is configured using a configuration file where its inbound and outbound streams are defined. The configuration file used in LONE is `flvplayback.lua`, and its relevant content can be seen in Listing 5.

```
acceptors =
  {
    {
      ip="0.0.0.0",
      port=1935,
      protocol="inboundRtmp"
    }
  },
externalStreams =
  {
    {
      uri="rtmp://flash.oit.duke.edu/vod/MP4:test/brunswick.m4v",
      localStreamName="test",
      forceTcp=true
    }
  },
```

Listing 5: Snippet of CRTMP flvplayback.lua Configuration

`Acceptors` define inbound- and `externalStreams` outbound connections to CRTMP. Inbound defines which ports users can connect to and `externalStreams` define the source of external input. The `externalSteam` seen in listing 5 has a video file on an external server as input, and gives the outbound stream the name `test`. This test-source was used to test if CRTMP was running correctly. To view the stream, the program Video Lan Client *(VLC)*, was used, as it has the same capabilities as flash in regard to streaming RTMP.

A connection is established with a URL-address of the format seen in Listing 6, where `server address` is the address of the CRTMP

server, `application` is the application where CRTMP lookup e.g. live or media, and `streamName` is the name of the stream.

```
1  rtmp://[server address]/[application]/[streamName]
```

Listing 6: CRTMP URL Format

When a user connects to CRTMP and the `application` is live, CRTMP will try to find a live stream that correspond to the link name. If no such live streams is found, CRTMP will look in the media folder and try to find file streams. If no file stream is found, CRTMP will wait for the live stream `streamName`.

CRTMP has supports user authentication, meaning access to streams can be limited to specific users. This has however not been implemented in LONE

4.1.3 *Issues with PaVE Headers*

The documentation for the PaVE headers can be found at [13, page. 59-60]. As described in Section 4.1.1, Flash applications are unable to interpret the PaVE headers. As a result of this, they are unable to display the video stream. These headers must therefore be removed or replaced with another header to make the stream readable. The implemented solution removes them with the `paveparse` plugin, described in Section 4.1.1. Removing the headers leaves raw video data with no information about how to interpret it. As a result of this, the Flash application, VLC, or any other video players is unable to interpret the video feed. An illustration of the PaVE header problem can be seen in Figure 22.

The PaVE headers result in a situation that makes displaying the video stream in Flash impossible, as the video feed cannot be interpreted by a Flash application neither with or without them.

4.1.4 *Testing with a test-input source*

The designed solution would work if the drones' video feed used standard headers. The issue is illustrated in Figure 23. The goal is to get the stream from the drone, through GStreamer and CRTMP to a Flash application. The implemented solution is capable of getting the video from the drone to CRTMP, but not capable of generating an output stream readable by Flash. As illustrated by 3. in Figure 23 the setup can forward a stream readable by Flash. This can be documented by using a different video source as illustrated by 4. For this

Frame with PaVE elements

| Header |
| Size |
| timestamp |
| frame |
| etc. |

Frame without PaVE elements

| ~~Header~~ |
| Size |
| timestamp |
| frame |
| etc. |

Figure 22: Illustration of videoframes with and without PaVe headers.

purpose the GStreamer plugin `videotestsrc` is used. It creates a test video stream as seen in Figure **??**, consisting of raw video data.

1.   Drone → GStreamer → CRTMP → Flash

2.   Drone → GStreamer → CRTMP

3.   GStreamer → CRTMP → Flash

4.   videotestsource → GStreamer → CRTMP → Flash
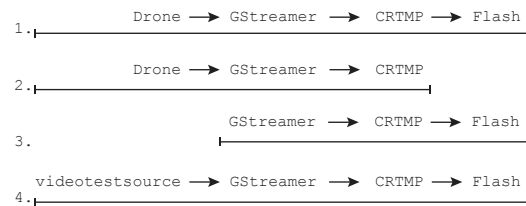
Figure 23: The Streaming

A `testvideosrc` pipeline can be seen in Listing 7

```
1  videotestsrc ! queue ! x264enc ! queue ! flvmux ! queue ! rtmpsink
       location='rtmp://0.0.0.0/live/myStream'
```

Listing 7: GStreamer Pipeline using videotestsrc

If this test-source is used instead of the drones' video feed, it is possible to display the video feed send by GStreamer.

This solution can be seen in figure 25.

4.1.5 *Implemented streaming solution*

The streaming setup used in LONE uses only GStreamer. One pipeline that reads the drones' video feed and serves as a server, denoted *SP*,
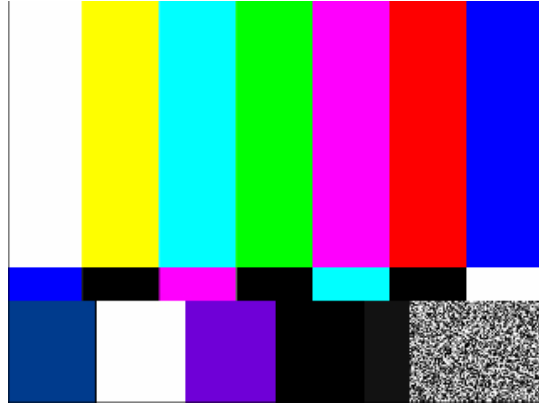
Figure 24: The test video source played with a Flash application.
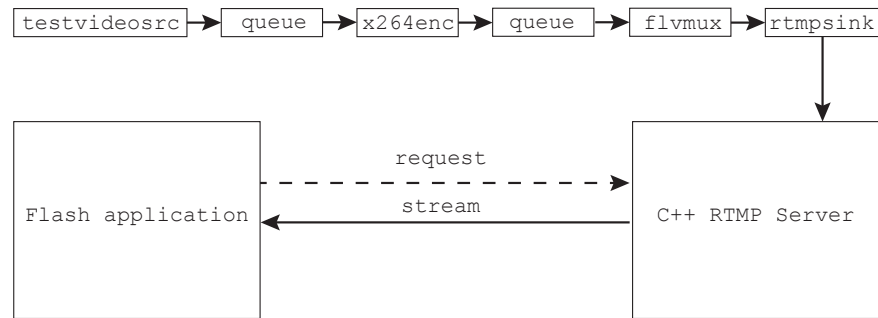
Figure 25: Illustration of the working solution with use of the video test source.

and a client pipeline, denoted *CP*, for displaying the video feed on the client side. The setup can be seen in Figure 26.
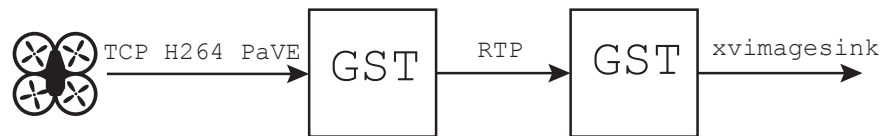
Figure 26: Illustration of the current solution with xvimagesink.

SP reads the drones' video feed and multicasts it using the Real-time Transport Protocol (RTP). RTP is a standardized protocol for sending audio and video packages over an IP network. RTMP is not used, as it is a protocol made specifically for Flash. For a GStreamer only solution, RTP is better choice as it is simpler to set up, since it does not have to include settings for the Flash application. RTP packages are sent via UDP. *SP* uses the `tcpclientsrc` and `paveparse` plugins used described in Section 4.1.1.

Video data send via RTP must be pay-loaded, as a **video frame!** (**video frame!**) is larger than the maximum size allowed size of a UDP

packet. Pay-loading a video stream means encapsulating it into a specific format. Pay-loading adds the RTP header to the data packages which encapsulates the pay-loaded data. GStreamer has a set of RTP pay- and depayloaders for each video format it supports. The plugin used in this pipeline is `rtph264pay`.

GStreamer does not have a dedicated RTP sink. As RTP packages are sent via UDP, the `udpsink` is used. `udpsink` has two parameters, `host` and `port`, and a set of optional flags with default values. The flag `auto-multicast` must be set to true to broadcast the stream globally. Accordingly the host is set to the multicast IP `224.1.1.1`. The port is set to `5123`.

The complete *SP* can be seen in Listing 8:

```
1  tcpclientsrc host=192.168.1.1 port=5555 ! paveparse ! rtph264pay !
      udpsink host=224.1.1.1 port=5123 auto-multicast=true
```

Listing 8: rtmpsink setup

CP uses the `udpsrc` element to read the video stream. `udpsrc` parameters are identical to those of `udpsink`. In order to decode the stream, an additional parameter named `caps` must be set. `caps` are used to describe metadata about the incoming video stream and contains information such as encoding, the protocol it is being sent using, **framerate!** (framerate!), etc. The caps are generated by *SP*. The `udpsrc` element of *CP* can be seen in Listing 9.

```
1  udpsrc uri=rtp://XXX.XXX.XXX.XXX port=5123 caps = "application/x-rtp,
      media=(string)video, clock-rate=(int)90000, encoding-name=(string)
      H264, sprop-parameter-sets=(string)\"Z01AFeygoP2AiAAAAwALuaygAHixbLA
      \\=\\,aOvssg\\=\\=\", payload=(int)96, ssrc=(uint)1171155755, clock-
      base=(uint)868988588, seqnum-base=(uint)65233"
```

Listing 9: udpsrc Setup

The next two steps of the pipeline is depayload the received data and decode it to raw video data. The depayloading is done with the `rtph264depay` plugin and the decoding with the `ffdec_h264` plugin. Following these two steps, the video data is reassembled after the transfer and been decoded. The last plugin used is `xvimagesink` which displays the video stream to the user. `xvimagesink` has two flags used to remove delay on the stream named `sync` and `async` which are both set to true. The complete *CP* can be seen in Listing 10.

```
1 udpsrc uri=rtp://XXX.XXX.XXX.XXX port=5123 caps = "application/x-rtp,
      media=(string)video, clock-rate=(int)90000, encoding-name=(string)
      H264, sprop-parameter-sets=(string)\"Z01AFeygoP2AiAAAAwALuaygAHixbLA
      \\=\\,aOvssg\\=\\=\", payload=(int)96, ssrc=(uint)1171155755, clock-
      base=(uint)868988588, seqnum-base=(uint)65233" ! rtph264depay !
      ffdec_h264 ! xvimagesink sync=false async=false
```

Listing 10: Client Pipeline

4.2   WEB-INTERFACE

The web-interface is implemented using Rails as described in Section 3.11.2. This framework enforces a number of standards that is followed in the implementation of the web-interface. The most significant is CRUD, mentioned in Section 3.6. The web-interface is implemented using the design mentioned in Section 3.6 and Appendix D.

A number of Rails-specific tools, named *Gems* [8], which are plugins for Rails written by the community, provides us with a number of tools that we can use and build our own application on. Five different Gems are used in LONE. These are:

- mysql2

- daemons-rails

- eventmachine

- sht_rails

- swfobject-rails

LONE uses a relational MySQL database to store all informations in. Ruby and Rails is not able to connect with MySQL out of the box. Therefore the *mysql2* [26] gem is is used, which provides a simple interface for Ruby to connect with a MySQL database.

The *daemons-rails* [16] is used for the daemon running in *M*. It accepts incoming requests from *S*'s and uses the models in the web-application to interact with these. To make sure that the daemon can use these, the daemons-rails gem is used.

The *eventmachine* Gem [14] is used to allow our web-application to easily interact with other programs using TCP/IP. The daemon running on *M* also uses this to communicate with *S*'s when exchanging informations regarding session keys. Whenever a user requests access to a drone, the request of a session key is saved in the database. The daemon running on *M* takes this request and uses the eventmachine to connect to the *S* associated with the drone that the user wishes to interact with, in order to sync session keys and verify that the connection between the user and drone is valid.

The *SHT_Rails* Gem *Shared Handlebars Template for Rails* [33] is used in the view of the web-interface and enables the application to render a view both as HTML and JSON, making it easier to integrate AJAX functionalities. This is used in the view, where AJAX is implemented as described in Section 3.11.3

The *SWFObject* Gem [23] allows us to easily display a swf flash object in the view of LONE, which is used when a user is interacting with a drone by either viewing its video feed or sending new commands to it.

All of the listed Gems are plugins written by third party that provides some functionality that we need somewhere in LONE.

TESTING

This chapter will cover the testing of LONE, how it was designed and performed. LONE was subjected to tests during the development phase. Following completion of the development a formal test of the functionality was conducted.

5.1 APPROACH

There exist two testing approaches: Black-box testing and White-box testing, which each have a dynamic and static methods as listed below.[1]

- Black-box testing
    - Static – Inspection and review of specifications i.e.
    - Dynamic – Acceptance tests.

- White-box testing
    - Static – Code inspection and review
    - Dynamic – Unit-testing

Two testing approaches is used in LONE, static White-box testing and dynamic Black-box testing. Static White-box testing was conducted continuously throughout the development. This was a result of pair programming being a part of the development method, as described in Section 2.3. Pair programming is code review as the code is being written[2]. Testing following completion of development has been limited to dynamic black-box testing. We have chosen to only use dynamic black-box testing, since it is closely related to our previous approach using use cases and defining acceptance tests seen in Section 2.5, and as LONE is only a proof-of-concept.

If LONE should go from proof-of-concept to deployment more extensive testing is needed. Unit-testing should be applied to ensure the functionality in LONE is fit to use. The two reasons we have chosen not to use unit-tests but reside to acceptance tests are that: 1) Setting up and executing unit tests is very time consuming and 2) LONE is a proof-of-concept and therefore acceptance-tests are deemed sufficient.

---

1 FiXme Fatal: source til testing bog
2 FiXme Fatal: source

The acceptance tests are defined in Figure 8 and Figure 9. They are derived from the use cases seen in Figure 6.All of the tests are manually performed by two teams of two persons each. The results are logged in Appendix E.

5.2   TEST REPORT

The tests that were successful and accepted can be seen in the appendix along with the other tests. One can expect that those tests that were accepted are fully implemented in LONE, are functional and ready to use. Those test that failed will be covered in this section.

ID	Use Case ID	Status	Description
2	2	Not passed.	Not implemented yet.
5	5	Not passed.	Not implemented yet.
6	6	Not passed.	The user is not capable of changing the name of a drone, as there is no name-field. The only identification is a hard-coded serial number.
13	10	Not passed.	Not implemented yet.
14	10	Not passed.	Not implemented yet.
22	13	Not passed.	Not implemented yet.
23	13	Not passed.	Not implemented yet.
28	17	Not passed.	Not implemented yet.

Figure 27: Acceptance tests results

The acceptance test results leads to the following three conclusions:

- The primary functions of LONE are implemented and working.

- However, some of the functionality can be improved and made more user-friendly.

- The failed acceptance tests listed in Table 27 are not needed for a proof-of-concept solution, but they are needed before LONE can be considered ready-to-market.

EPILOGUE

This chapter rounds up this report. First a discussion and reflection will evaluate the work and result of LONE. Then a conclusion that answers the problem statement found in Section 1.5. And finally a section on future work, that elaborates on how this project can be improved in the future.

6.1 DISCUSSION & REFLECTION

During the design and implementation of LONE we have made a series of choices that has affected the outcome of the project. This section will elaborate on these choices and the impact they have had on the final product.

One of the first decisions we made was that we will use the programming language Ruby and the development framework Ruby on Rails. We knew that Rails came with a lot of functionality, such as using the MVC model and Active Relations and other features that helped us use out time as efficient as possible when we were developing. On the other hand we had little to none previous experience with Ruby or Rails, so the time saved on the built-in feature of Rails was spent on learning the programming language instead. We believe this was a wise decision. One of the preliminary boundaries of this project was, that it is a student project. We therefore believe that it is a good thing, that we got to learn a new programming language. We may have been able to implement this project faster if we were using a programming language that we were more familiar with, such as PHP, but we believe that the learning process is worth more than the time we might have saved.

Looking at the final product, LONE, we are satisfied with the results. There are, however, elements in it that could be improved. Some are important to look at while others are less important until the product should be made ready for market.

Whenever a client (Browser *B*) is trying to access a Drone *D*, it must receive a session key from *M* generated on *S*. This key is transfered unencrypted via regular HTTP. This means that the key can be caught by third party if he can sniff the network packages sent on the network of either *S* or *B*. This has no impact on the functionality of the

53

system, but it is a security concern that should be looked at before LONE is ready to market.

Another potential security issue is the registration of new Slaves *S*. There is no authentication of new *S* in the current implementation, meaning that any computer can connect to *M*. If they provide a valid serial ID when they connect the first time, they will be saved in the system as an active and valid *S*. This can lead to different security issues, but as the previous security issue it has no significant impact no the functionality of LONE. It must, however, be fixed before LONE can be seen as ready to market.

A lot of our time was spent on streaming the video feed from the drone to the clients browser. Transporting the video feed from the drone to the slave *S* was no problem, for this purpose Gstreamer was used and set up with appropriate options on *S* to fetch the feed from the drone. Displaying the video feed in the flash application in the clients browser did not work because of the lack of PaVE headers as described in Section 4.1.3. A lot of time was spent on this, as displaying the video feed from the drone is one of the important features of LONE. Instead we ended up implementing the solution described in Section 4.1.5, which works (sort of) but in no way is an ideal implementation. It requires the client to run a separate application on his computer to view the video stream, and have his computer focused on the flash application in the browser at the same time to be able to send commands to the drone.

This is not a satisfying solution. However, it works as a proof of concept implementation. It is possible to both view the video feed from the drone and send commands to it that it will react too. If LONE was suppose to be ready to market now, a solution where the video feed is presented in the browser, preferably in the same flash application that takes the flight commands, is a requirement.

A lot of time was spent trying to implement the first solution, as we are aware of the importance if this, given the case that LONE should be ready to market. However, when we came to the conclusion that we could not get it to work within the given time of this project, the secondary solution was implemented, which – in our eyes – is better than no video feed and still makes a proof of concept. We do not consider the time wasted, as it still provided the project with a lot of valuable knowledge about streaming and how to implement such solutions.

During the design and implementation of LONE we have used work sheets to document our thoughts and work, to make it easier when we had to write this report. This also took a lot of time, but is considered a good investment as it provided useful resources in

writing this report.

LONE is not a perfect system, and it is not ready to market. However, there are elements in it that we are very proud of, and that we think is good work.

- The architecture that binds a drone and a user together, using a browser, slave and master-server.

- The object-model of the structure of the web application hosted on the master.

- The communication with the drone that allows us to control its movements via a flash application in the clients browser.

These are the most basic elements in LONEś foundation, that we are proud of and believe are well implemented.

6.2 CONCLUSION

The purpose of this report is to describe how we developed a solution to the following problem:

How can drone technology be applied in a scalable web application to improve the efficiency and cost-efficiency of remote video surveillance of large outdoor areas?

Before developing a solution to the problem it was necessary to define the problem domain. The problem domain was defined through an analysis of existing solutions and interviews with a security company. From this definition of the problem domain we identified the issues as: Scalability, wireless short-distance and wired long-distance communication and streaming, and permissions- and access control.

Use cases were created based on these issues, and the functionality of LONE was then derived from the use cases. The design was based on this functionality and the use cases. LONE was both designed and implemented using an iterative process, where a combination of SCRUM and XP was applied. Each iteration contained planning, designing, implementation and testing. This report shows the final results of the design and implementation of all the iterations.

The *Master* and *Slave* architecture makes LONE able to scale in regards to drones. The *Master* is a server responsible for the web application, the database and establishing connections to slaves. A *Slave* is a server associated with one drone. All communication with this drone goes through the slave that it is connected to.

LONE supports that a user communicates directly with a *Slave* without going through *Master*, thus limiting the workload of *Master*. The web application on Master is designed and implemented to allows the amount of users to grow.

Wireless communication is used to communicate with the drone. Since the drone is remotely controlled, this communication includes both control commands and video streaming. A streaming solution was designed using the tools GStreamer and C++ RTMP Server. Due to the format of the drones' video feed, it was not possible to implement a working streaming solution into the web application as described in the use cases. Instead, this version of LONE has another type of video streaming implemented that requires a third party video player installed on the client-side to show the video stream from a drone.

Users gain access to LONE through the privilege concept which is implemented into the web application. This concept is designed so that no user has access to any functionality, and in order to gain access they have to be granted privileges. Since the user communicates through a *Slave* when communicating with a drone, it is necessary to implement access control on the *Slave* too. This is done by enforcing that all communication to a *Slave* happens through a valid session. A valid session is achieved using a *session key*, which can be obtained in the web application on the Master, if the user has the required privileges. A session key is a unique key that verifies that the session between a user and a *Slave* is valid, and that the user has the right permissions to communicate with this *Slave*.

Based on the use cases, a number of acceptance tests were created which the testing of LONE was based upon. Only acceptance tests were run in LONE. These were run to ensure that all of the required functionality were present and working. They did, however, show that some of the functionality in the acceptance tests, derived from the use cases, is not working or is not implemented. We do not consider any of this functionality to be critical for a proof-of-concept version of LONE, but they have all been listed in "future work" for later implementation. Most of the acceptance tests were accepted, showing that the functionality that was designed is also implemented correctly into LONE.

LONE in its current form is not ready-to-market. It is, however, a proof-of-concept solution showing how drone technology can be applied in a scalable web application to improve the efficiency and cost-efficiency of remote video surveillance of large outdoor areas.

6.3 FUTURE WORK

LONE in its current state is, as mentioned previously, a proof of concept. It is not ready to market yet. A couple of things has to be fixed or implemented in order to make it fully ready to market. These are considered future work and are covered in this section.

A number of the accepts testes were not accepted when the tests were run. These can be seen in Section 5.2. All of the tests that were not accepted must be implemented / fixed before LONE is ready to market.

As described in Section 5.1 only dynamic black-box testing (acceptance tests) were run on LONE to verify that it is implemented as planned and designed. When moving the project from a proof of concept state to a ready to market state, we believe that dynamic white box testing (unit testing) is necessary in order to ensure that the system works 100% correct at all times, given any input.

As described in Section 6.1, the streaming implemented in LONE now is not a solution that can be shipped to the market. The solution that was originally planned for implementation – or a similar solution where the video feed is displayed in the clients browser – must be implemented into LONE before it is ready to market.

Those are the things that needs to be fixed or improved in the project as it looks now. Besides that there are two things that qualifies as future work to make LONE ready to market.

- Radio-controlled drones – It is not an optimal solution that the drones broadcast their own WIFI network that S has to connect to in order to be able to communicate with the drone. Instead a desirable solution is that the drone can receive a given radio signal. This gives two advantages. 1) It has a much larger range than then WIFI signal that the AR Drone, used in this project, has. And 2) it will be better if the drone can connect to a broadcasted signal rather than it is the one broadcasting the signal, as we will have better control with a signal that the S broadcasts itself.

- Better hardware – The current drones has limited fly time due to the batteries. This needs to be improved. The WIFI network card that we are using in S is also of low quality. It has low range and we have seen the driver crash when the connection is interrupted (e.g. by cutting the power to the drone). This hardware needs to be improved before LONE can be used in production.

Once this list of fixes and improvements has been implemented, LONE is getting close to a ready to market state. After completing an intense testing session, of course.

Part I

APPENDIX

INTERVIEW WITH LYTZEN IT

The interview were setup with Lytzen IT via email and they were the only alarm company to contact us back.

Lytzen IT were presented by both Søren Ole Søndergaard and Jesper Toft. Søren is one of the partners that owns Lytzen IT and Jesper is their alarm and security specialist. The meeting found place at Lytzen AS and Lytzen IT's headquarters in Hjørring, Friday the 12 December.

In the meeting the project were presented and the idea about drone surveillance. What the thought was and what the system could solve of problems in todays world. An example was surveillance of AAU buildings where there are install contacts on every window in this way they can tricker an alarm. The alarm will only go off in a certain time interval of the day if it is a workday and all the time if it is in the weekends. One of the problems is if a student accidently opens a window and the alarm goes of a guard must come and check it out which is very costly. Here the drone could be placed inside the building and flown around to check alarms before sending a guard.

Lytzen IT had some cases where they either have used a lot of small cameras or one big dome camera to make a CCTV, Closed-Circuit TeleVision. A dome camera is a camera which can turn 360° and in the big models have a great optic zoom. Because of all these features these dome cameras can surveillance a big area but there is some short comings. They are expensive, they have to be placed high up to be able to see everything and if there are objects they cant see "behind" them. For out the cameras being very expensive the cables for these cameras are even more so. They need to be shielded against weather and being in the ground. They need to be laid in a certain depth. In large outdoor areas the drones are probably more suited than the stationary cameras. They can go almost everywhere and are not as expensive as the big dome camera or a lot of small cameras with all the cables needed. But the drone system presented also have some short comings. Firstly it is using the global Internet. They think as CCTV is a closed-circuit our drone stream and communication should be the same. Lytzen also pointed out that the station where the drone needs to charge needs to have some technology that ensures the drone can just land and then charge without any human interaction. The "charge to fly time ratio" should be better before it can be used in a real case. Another thing was that the drone maybe should use radio communication instead of WIFI because it would both give a more secure line and make the range of the drone higher. Also they think that the drone should be automated more so it is not dependent

61

on a pilot all the time. But they still thought that it will be the future in 5 to 10 years. Surveillance is about getting evidence for the police, insurance company and keeping the evidence safe. The drone can also have a intimidating effect. This is why they think that drone surveillance is the future. The last problem that Lytzen pointed out was the law. In Denmark you may not record any public places. This could be troublesome when using a drone. But Lytzen IT thinks this is likely to change as Denmark is moving against a total surveillance society.

B

AR DRONE TECHNICAL SPECIFICATION

The AR Drone has its own wireless network that one must connect to in order to control it. It will always have the ip `192.168.1.1`. Communication with the drone is done using this IP through three ports:

- 5554: The drone sends out information on this port to its clients. This information is e.g. its status(*flying*, *landed* etc.), and its speed. The information send from the drone is reffered to as *navdata*.

- 5555: The drone sends out a video stream on this port via TCP.

- 5556: The drone is controlled and configured by sending *AT commands* to it on this port via UDP.

AT commands are text strings, which refer to a specific function call in the drones library, as such parameters are send to the drone along with the command. The most important AT commands are:

- `AT*FTRIM` - configures the drones notion of a horizontal plane. Drone must be on ground when the command is send.

- `AT*REF` - Take off, land, and emergency stop command. Takes an integer value as argument.

- `AT*PCMD` - Controls the drone. Takes 5 arguments: a flag, and four integer values which defines the drones movement. When a `PCMD` command is received the drone sets its control values to those received and then resets them. For consistent movements the drone must receive at least 30 packets per second.

The video stream send by the drone on port 5555 is encoded in the H.264 format. Network video streaming is done by sending the video frames individually. Each video frame has a custom header containing metadata on the frame. The stream send by the drone uses a unique header PaVE *(Parrot Video Encapsulation.*

63

MODEL FUNCITONS

Function	Description
privileges	returns all privileges associated with a company

Figure 28: Company Model

Function	Description
type	returns the type of the privilege
type=	Assings a type to the privilege, raises an exception if the assigned type does not exist

Figure 29: Privilege Model

Function	Description
unlink	unlinks the drone from its associated company
privileges	returns the privileges associated witht the drone

Figure 30: Drone Model

65

Function	Description
`check_no_other_sessions_for_drone_exists`	Checks if a session exists for a drone and raises an exception if this is the case

Figure 31: Session Model

Function	Description
`privileges`	Returns the privileges of the role

Figure 32: Role Model

Function	Description
`authenticate`	Have an e-mail and a password as parameters. Returns the user object if it is a valid user, or false if it is not.
`make_salt`	Accepts a parameter to be used in generation of a salt and returns it
`hash_with_salt`	Have a password and a salt as parameters. Returns the password hashed with salt.
`password_match?`	Has a password as a parameter. Returns true if the password with the user objects salt matched the stored.
`has_privilege?`	Has a privilege as a parameter and returns true or false based on whether or not the user has the privilege
`privileges`	Returns the privileges of the role

Figure 33: User Model

CONTROLLER ACTIONS

Action	Description
login	Checks if user is logged in or redirects user to login page.
attempt_login	Attempts to log in the user, if it fails shows the login page again along with an error. If the user is logged in the user is redirect to the application and is shown a message saying the login was successful.
logout	Performs a logout for the user, and redirects him to the login page.

Figure 34: Access Controller

Action	Description
CRUD	Create, Read, Update, Delete the Model

Figure 35: Affiliate Privilege Controller

Action	Description
CRUD	Create, Read, Update, Delete the Model
users	Returns two arrays: all users within the company and all users not in the company.
companies_users	Adds/removes a user to/from a company depending on the action.
privileges	Returns two arrays: all users within the company and all users not in the company's primary role.
companies_privileges	Adds/removes a privilege to/from a company's primary role depending on the action.
drones	Returns the company.
companies_drones	Adds a drone to a company.
roles	Returns the company.
companies_roles	Creates and adds or deletes and removes a role to/from a company.

Figure 36: Company Controller

Action	Description
CRUD	Create, Read, Update, Delete the Model
get_information	Gets information of a drone given a name of the drone, also returns the companies of the current user.
link_drone_to_company	Links a drone to a company.

Figure 37: Drone Controller

Action	Description
CRUD	Create, Read, Update, Delete the Model
search	Searches for privileges within the companies of a role or user.

Figure 38: Privilege Controller

Action	Description
CRUD	Create, Read, Update, Delete the Model
add_privileges	Adds affiliate privileges to a role.
remove_privileges	Removes affiliate privileges from a role.
add_users	Adds users to a role.
remove_users	Removes users from a role.

Figure 39: Role Controller

Action	Description
CRUD	Create, Read, Update, Delete the Model
search	Searches by full name for a user within the companies of a role.

Figure 40: User Controller

E

ACCEPTANCE TEST RESULTS

E.1 ACCEPTANCE TEST RUN

ID	Use Case ID	Status	Description
1	1	Accepted.	The user is provided with a username and password form, that gives visual feedback based on the users action (logging in, failed attempt).
2	2	Not passed.	The user, after performing a valid login, is only shown content according to his privileges.
3	3	Accepted.	The user with rights is shown a page with an interface that allows him to pilot a specific drone.
4	4	Accepted.	The user with rights is shown a page with a window that enables him to see the video feed of a specific drone.
5	5	Not passed.	The user can successfully grant or revoke a privilege to another user.
6	6	Not passed.	The user is able to change the name of the drone.
7	7	Accepted.	The user is able to link a drone to a company.
8	7	Accepted.	The user is able to unlink a drone from a company.
9	8	Accepted.	The user is presented with a concise list of available drones.
10	9	Accepted.	The user is able to press a link to logout of the system.
11	10	Accepted.	The user with rights is able to create a new user via an interface.
12	10	Accepted.	The user with rights is able to edit an existing user via an interface.
13	10	Not passed.	The user with rights is able to deactivate an existing user via an interface.

Figure 41: Acceptance Tests 1

ID	Use Case ID	Status	Description
14	10	Not passed.	The user with rights is able to activate an existing user via an interface.
15	11	Accepted.	The user is able to create a company via an interface.
16	11	Accepted.	The user with rights is able to remove a company.
17	12	Accepted.	The user is able to add users to the company.
18	12	Accepted.	The user is able to remove users from the company.
19	12	Accepted.	The user is able to add new roles to the company.
20	12	Accepted.	The user is able to edit existing roles in the company.
21	12	Accepted.	The user is able to remove existing roles from a company.
22	13	Not passed.	The user with rights is able to grant his own privileges to another user within the same company.
23	13	Not passed.	The user with rights is able to remove privileges from other users within the same company that he is able to grant them.
24	15	Accepted.	The user with rights can add privileges to the role.
25	15	Accepted.	The user with rights can remove privileges from the role.
26	16	Accepted.	The user with rights can add users to roles.
27	16	Accepted.	The user with rights can remove users from roles.
28	17	Not passed.	The user is not able to pilot a drone that is already being piloted.

Figure 42: Acceptance tests 2

OBJECTS & RICH RELATIONSHIPS

Each attribute named `id` is a unique integer relative to the object.

F.1 OBJECTS

Attribute	Description
id	See definition.
first_name	The user's first name.
last_name	The user's last name.
email	The email of the user. Must be unique.
salt	A hashed string based on email and time.
hashed_password	A hashed password based on the salt and an inputted string.
password	The password in clear text. Only temporary, thus not stored in the database.

Figure 43: User Object

Attribute	Description
id	See definition.
ip	The IPv4 of the drone's slave.
location	The location of the drone's slave.
name	A unique string that represents the drone, which is predefined on the drone's slave.
description	Optional string to describe the drone.
company_id	A reference to the company object.

Figure 44: Drone Object

F.2 RICH RELATIONSHIPS

Attribute	Description
id	See definition.
name	The title of the company.
owner	The id of the user owning the company.

Figure 45: Company Object

Attribute	Description
id	See definition.
title	The name of the role.
level_type	An integer representing a non-hierarchy level of roles.

Figure 46: Role Object

Attribute	Description
id	See definition.
identifier	A string that combined with instance_type is unique.
description	A description of the privilege.
instance_type	An integer representing the type of privilege, such as global, company, and drone.

Figure 47: Privilege Object

Attribute	Description
id	See definition.
privilege_id	A reference to the privilege id.
affiliate	The id of the instance type specified by the privilege.

Figure 48: Affiliate Privilege Object

Attribute	Description
`id`	See definition.
`drone_id`	A reference to the drone.
`user_id`	A reference to the user.
`session_key`	The session key for the user and drone combination.

Figure 49: Session Object

Attribute	Description
`id`	See definition.
`drone_id`	A reference to the drone of which to get session for.

Figure 50: Session Key Task Object

Attribute	Description
`id`	See definition.
`user_id`	A reference to the user.
`affiliate_privilege_id`	A reference to the affiliate privilege.
`flag`	A flag that indicates if the privilege should be granted or revoked.

Figure 51: User Privileges Relationships

BIBLIOGRAPHY

[1] Ffmpeg - project description. 2007. URL http://ffmpeg.org/.

[2] Ffserver - how does it work. December 2012. URL http://ffmpeg.org/ffserver.html#How-does-it-work_003f.

[3] Adobe. Actionscript 3.0 reference for the adobe flash platform - netstream - as3. 15/11/2012. URL http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/NetStream.html.

[4] Adobe. Real time messaging protocol (rtmp) specification. 2009. URL http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf.

[5] Sagar G Arlekar. The role of ajax in enhancing the user experience on the web. 2006. URL http://www.roseindia.net/ajax/ajax-user-interface.shtml.

[6] C2.com. Create read update delete. URL http://c2.com/cgi/wiki?CreateReadUpdateDelete.

[7] Stephen Chapman. What is javascript? URL http://javascript.about.com/od/reference/p/javascript.htm.

[8] Rails Community. Gems for ruby on rails. URL http://rubygems.org/gems/rails.

[9] Douglas Crockford. Json. 2002. URL http://www.json.org.

[10] Debian. Debian – the universal operating system. April 2012. URL http://www.debian.org.

[11] Alexis Deveria. Can i use the html5 video element. 2012. URL http://caniuse.com/video.

[12] Python Documentation. The http protocol is stateless. *Python*. URL http://pythonweb.org/projects/webmodules/doc/0.5.3/html_multipage/lib/node145.html.

[13] Pierre Eline et al. Ar drone developer guide. *AR Drone*, 2009. URL https://projects.ardrone.org/attachments/download/434/ARDrone_SDK_2_0.tar.gz.

[14] Aman Gupta Francis Cianfrocca. Eventmachine gem for rails. URL http://rubygems.org/gems/eventmachine.

[15] GStreamer. Gstreamer plugin – queue. URL http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-plugins/html/gstreamer-plugins-queue.html.

[16] David Heinemeier Hansson. Daemons gem for rails. . URL http://rubygems.org/gems/rails.

[17] David Heinemeier Hansson. Getting started with rails. . URL http://guides.rubyonrails.org/getting_started.html.

[18] Rational Software Microsoft Hewlett-Packard Oracle Sterling Software MCI Systemhouse Unisys ICON Computing IntelliCorp i-Logix IBM ObjecTime latinum Technology Ptech Taskon Reich Technologies Softeam. Uml notation guide. 05/08/1997. URL http://www.cse.wustl.edu/~kjg/cse132/forms/UML%20notation%20guide.pdf.

[19] Justitsministeriet. flvmux. *GStreamer Good Plugins 1.0 Plugins Reference Manual* -. URL http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-flvmux.html.

[20] Justitsministeriet. BekendtgÃžrelse af lov om tv-overvÃ¥gning. *Retsplejeloven*, 2007. URL https://www.retsinformation.dk/Forms/r0710.aspx?id=105112.

[21] Dardo Kleiner. Ar drone 2.0 video data without the sdk. 14/8/2012. URL https://projects.ardrone.org/boards/1/topics/show/4282#message-4725.

[22] McAfee Labs. Password policy - length vs. complexity. 2/11/2007. URL http://blogs.mcafee.com/mcafee-labs/password-policy-length-vs-complexity.

[23] Corin Langosch. Swf object gem for rails. URL http://rubygems.org/gems/swfobject-rails.

[24] Craig Larman. *Agile & Iterative Development*. ADDISON WESLEY, first edition, 2004. ISBN 0-13-111155-8.

[25] Paul Lewis. You're being watched: there's one cctv camera for every 32 people in uk. *The Guardian*, March 2011. URL http://www.guardian.co.uk/uk/2011/mar/02/cctv-cameras-watching-surveillance.

[26] Brian Lopez. Mysql gem for rails. URL http://rubygems.org/gems/mysql2.

[27] Microsoft. Model-view-controller. *MSDN*. URL http://msdn.microsoft.com/en-us/library/ff649643.aspx.

[28] Renee Puels. Tcom 598 independent study of telecommunications. 2006. URL http://teal.gmu.edu/telecom/publications/TCOM598-2006-Puels-UAVs.doc.

[29] Manoj Srivastava. Why debian. 05/31/2007. URL http://people.debian.org/~srivasta/talks/why_debian/talk.html.

[30] TechTarget. stateless. April 2005. URL http://whatis.techtarget.com/definition/statelessl.

[31] Dejan Todorovic. Gestalt principles. *Scholarpedia*. URL http://www.scholarpedia.org/article/Gestalt_principles.

[32] American Civil Liberties Union. What's wrong with public video surveillance? *ACLU*, 2002. URL http://www.aclu.org/technology-and-liberty/whats-wrong-public-video-surveillance.

[33] Alexey Vasiliev. Shared handlebars template gem for rails. URL http://rubygems.org/gems/sht_rails.

DECLARATION

Put your declaration here.

Aalborg, December 2012

_____

signature