LONE

LONE is Observation of Nondeterministic Environments

Department of Computer Science
Aalborg University
December 20th 2012

AALBORG UNIVERSITY
STUDENT REPORT

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East
http://www.cs.aau.dk

Title:

LONE — LONE is Observation of Nondeterministic Environments

Theme:

Internet Technology

Project Term:

P7, Fall 2012

Project Group:

SW701E12

Students:

Anders Eiler
Bjarke Hesthaven Søndergaard
Esben Pilgaard Møller
Rasmus Steiniche
Thomas Kobber Panum

Supervisor:

Lone Leth Thomsen

Copies: 6

Pages: 78

Finished: December 20th, 2012.

Synopsis:

Video surveillance is becoming more and more used around the world. The ways technology allow us to video surveil today is, however, quite expensive and has its limitations. This paper seeks to take advantage of modern technology to provide a new approach of video surveil large areas in a cost-efficient way. The goal is a scalable web-application that allows multiple users to control and view the video stream unmanned air-crafts to do remote and cost-efficient surveillance of large areas.

ABSTRACT

Video surveillance is becoming more and more used around the world. Governments use it to prevent criminal activities and terror, while the private sector use it to surveil their property and for gathering crime related evidence. The ways technology allows us to video surveil today is, however, quite expensive and has its limitations such as the amount of needed hardware to get a sufficient degree of surveillance and the installation costs of this equipment.

This paper seeks to take advantage of modern technology to provide a new approach of video surveil large areas in a cost-efficient way. A proof of concept solution that uses unmanned air-crafts with mounted cameras – drones – and a web-interface for user interacting based on a scalable infrastructure is designed and implemented. The goal is a scalable web-application that allows multiple users to control and view the video stream from such drones to do remote and cost-efficient surveillance of large areas.

The system that was developed in this student project is a proof of concept solution. It also shows that if better hardware is available, a solution that uses drones to surveil large our-door areas is possible to deploy.

PREFACE

We would like to thank our supervisor Lone Leth Thomsen for helping and supervising throughout the project.
Furthermore we would like to thank Dardo Kleiner for developing the GStreamer PaVE parser plugin for GStreamer.

QUOTATIONS are the words of another person along with a source. The source of the citation will either be in the text immediately before or after, or could potentially be incorporated into the quote as shown in the example below:

> *"This is an example of a quotation."*
>
> — X, p. Y

REFERENCES are references to sections, figures, code snippets, chapters or parts written elsewhere in the report. This could look like the following:

This is described in Section X.Y.

CODE EXAMPLES are written in a special environment so they are easy to read and recognize. Examples can be seen in Code snippet 1. Whenever there is a sequence of three dots ("...") in a code snippet, it means that we have omitted some content, which is not important in that specific context.

```
1  <?php
2
3  function print_numbers($from, $to) {
4
5      echo "Hello World! Here are the numbers from " . $from . " to " . $to
           ;
6      for($i = $from; $i <= $to; $i++) {
7          echo "\n" . $i;
8      }
9
10 }
11 print_numbers(1,100);
```

Listing 1: Code example of a hello world script written in PHP.

CONTENTS

ACRONYMS

GUI Graphical User Interface

Video Frame A coded still image in video technology

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

Framerate The frequency with which a new video frame is displayed in a video. Is often measured in Frames per second.

Rails Ruby on Rails

LONE LONE is Observation of Nondeterministic Environments

CRUD Create, Read, Update, and Delete

FIFO First In, First Out

AJAX Asynchronous JavaScript and XML

RTMP Real Time Messaging Protocol

CRTMP C++ RTMP Server

paveparse GStreamer PaVE parser

DoS Denial-of-service attack

SHTRails Shared Handlebars Template for Rails

VLC Video Lan Client

RTP Real-time Transport Protocol

MVC Model-View-Controller

1 INTRODUCTION

When a crime occurs documentation is often needed in order to convict the person(s) responsible and to receive payment from an insurance company for the loss suffered. Documentation can be eye witnesses, forensic evidence, or video surveillance. Eye witnesses or forensic evidence are unreliable sources for documentation, since it is not possible to guarantee they will be available. Documentation from 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 an optimal solution in its current form [38].

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 [28]. 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, but additionally due to the technical difficulties of video surveillance.

A typical video surveillance solution consists 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.

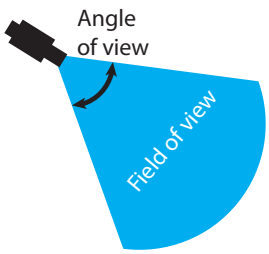

Figure 1: A camera's field of view.

An environment can contain physical objects which limits the camera's field of view further, see Figure 2. In small environments a

sufficient degree of surveillance can be achieved without too much difficulty or cost (based on the interview described in 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.:

> *" 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 camera's field of view is not a hard limit for what it can record, however only what is recorded within the field of view is of high enough quality to be used 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 both to cameras and 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 can be seen as 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 or rainy and the sun can blind a camera if it is improperly placed. These conditions make video surveillance of outdoor areas difficult. Both indoors and outdoors have some challenges that need to be faced, such as physical objects being moved around creating blind spots, etc.

1.2 EXISTING SOLUTIONS

This section will discuss some of the existing 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. In Figure 2 it can be seen how an object is placed in a camera's field of view, which creates a blind spot in an area that the camera would normally cover. 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.

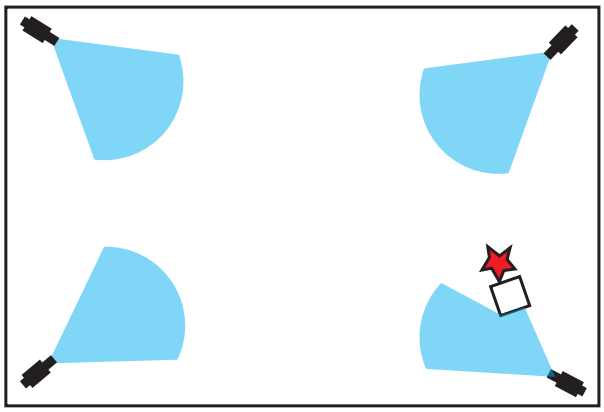

Figure 2: Example of a Regular Camera Setup.

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 has a dome camera with a long field of view positioned in the center. The perimeter is then divided into zones, by e.g. photo sensors. When a zone has 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 observing a specific subarea at a time as seen in Figure 4.

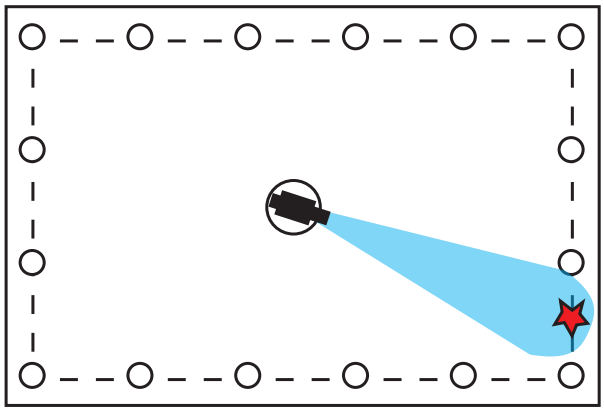

Figure 4: Example of a Dome Setup.

1.3 DRONE TECHNOLOGY

As described in Section 1.1 the problems with surveillance are 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 in an environment. This can be achieved with the use of drones. A drone in LONE is an unmanned aircraft which is remote controlled. Drones capable of moving 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. The need for cables can also be removed by using drones, as they can be controlled using wireless technologies such as WLAN [15], radio waves, or satellite [33].

1.4 WEB FOR SURVEILLANCE

With the development and standardization of network technologies, video surveillance is becoming digitized [34]. The digitization of video

surveillance is important as it, e.g. makes it faster to examine a video recording.Video recordings are examined faster as it is easier to skip through digital data compared to the analog data which would be stored on e.g. a VHS cassette. 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.

1.5 PROBLEM STATEMENT

Video surveillance in its current form is stationary, meaning cameras cannot move out of their mounted position. As a consequence of this, the amount of hardware required is proportional to the size of the area to be surveilled. This means that a large area is expensive to surveil, due to the costs of hardware. 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 drones remotely over long distances.
- How to provide video streaming of a drone's camera through a web application.
- How to make a web application scalable to support multiple drones and users.
- How the make a system accessible from remote locations in a secure manner.

With the preliminary problem statement and the following aspects considered the following problem statement were yielded::

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 developed as if it was a service offered by a company. Therefore the system, known as LONE, must be developed as

if it was to surveil several locations simultaneously. The hardware available is an AR Drone 2.0 Parrot, see Figure 5. It is not possible to develop a system usable in the real world using this hardware. This means the system will be developed to be a proof of concept.

Figure 5: AR Drone 2.0 Parrot.

2 ANALYSIS

In this chapter the problem domain for the problem statement defined in Section 1.5 is analyzed. The problem domain is video surveillance of large outdoor areas. In the analysis the potential issues with developing the proposed solution are identified and presented. Based on the analysis a set of requirements of the application is presented and based on this 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 handle this using a drone with a camera instead of stationary cameras. The solution is intended to be used commercially by different businesses that may each have several different areas or locations that needs to be surveilled.

Video surveillance using drones has a set of challenges which must be considered. In order to ensure that the drone performs the commands that it is sent, the connection to the drone have to be reliable. If the connection to the drone is lost the drone might crash, which must be avoided. The large outdoor areas require 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 using the web application as proposed in the problem statement. Furthermore the system must be scalable to include several locations, meaning several drones, and users simultaneously.

For video surveillance to be usable as documentation 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 in Denmark [23], where this project is developed.

This is not an issue for video surveillance with stationary cameras, as they are simply placed in locations that ensure they do not violate these laws. With movable cameras it needs to be ensured 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 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. This 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 application, as proposed in the problem statement.
- A drone must be controllable from the web application, as proposed in the problem statement.
- The system must support multiple drones and users.

A set of use cases have been derived from these requirements.

2.2 USE CASES

The functionality of the system will be defined through use cases, describing how a user is expected to interact with LONE. The use cases are derived from the requirements described in Section 2.1. 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. 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 are needed, such as controlling a drone.

The terms *User*, *Drone* and *Permission* in context of LONE are therefore introduced. *Users* are connected to *Drones* through *Permissions*.

An example can be an organization that has ten employees to take care of the security and has three drones in LONE. All ten employees must have access to all three drones.

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

The five objects user, drone, permission, company, and role, define the security and permissions of a user's interaction with LONE and are used in the use cases in Table 6. Use cases are refered to as UT_X where X is the id of the use case.

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 objects reflect elements in the problem domain which must be modelled in the system. 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 permission. As defined in UT_8 privileges grant access to functionality, meaning a *user* will not have access to anything by default. A *drone* refers to a physical drone. *Companies* are 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. UT_{15} defines a role as a 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.

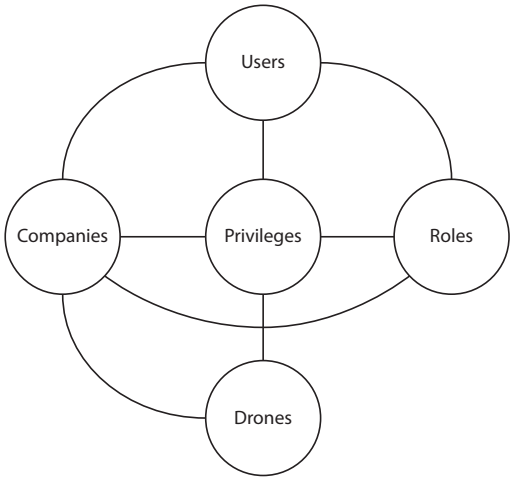

Figure 7: Element Diagram.

Since video surveillance is concerned with sensitive information, security in LONE is important. This is reflected in UT_1 and UT_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 objects 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.

2.3 DEVELOPMENT METHOD

The choice of development method is based on previous experience and which methods suit the project. The suited development method is determined by the circumstances under which the project is conducted. The project is conducted in a 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, since there are no uncertainties in regards to the required functionality of the system. The functionality defined by use cases does however contain unknown subjects such as video streaming, distributed computing, and communication with a drone. These unknown subjects create an uncertainty about how to implement the desired functionality. Another 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. Traditional developments methods are suited for projects where the solution domain is known and it is possible to create a upfront design. We deem this is not the case for this project, due to the uncertainty of the solution domain. The uncertainty in the solution domain is based on the unknown subjects mentioned. 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 an agile development method is better suited for this project. 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 [27].

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 standup meetings. The practices taken from XP are Pair-programming and test-driven development. The type of test used during test-driven development is acceptance test.

2.4 SYSTEM DEFINITION

LONE is a web service for surveilling large areas with cameras mounted on drones. The service will be accessible via a web browser. Each drone can only be controlled by one user at a time. Several users can view one drone's video feed simultaneously. 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. 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.

2.5 ACCEPTANCE TESTS

The development method used in this project makes use of acceptance tests, as described in 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 tests 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.

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.

3 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. 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 for 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 handles HTTP requests from users. The users HTTP requests are sent using a Browser, which will is denoted *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. DoS is an attempt to make a service unavailble, which is possible when a service is made available by a singleton server.

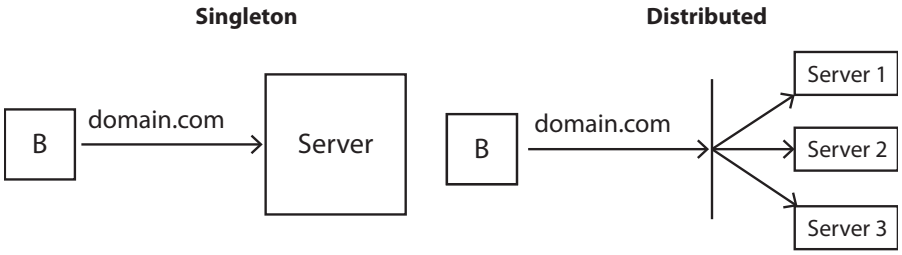

Figure 10: Illustration of a singleton and distributed server solution.

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 as data needs to be synchronized across several servers 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 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_3. In case the communication is lost the drone might end up crashing. The range limitations of antennas for digital wireless communication set a constraint for the availability of drones. Assuming there exists no antenna capable of covering the entire physical area of our problem domain, it is necessary to have a distributed antenna setup. This leads to the structure shown in Figure 11.

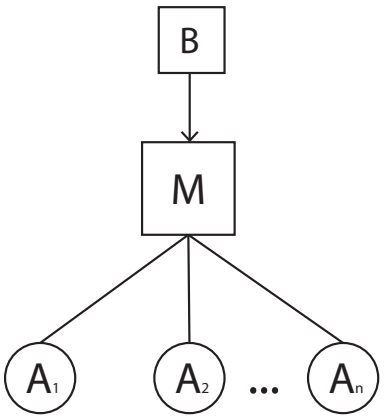

Figure 11: Antenna structure.

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 lost. Providing the antennas with processing power and opportunity to communicate directly with *B* would solve this issue. If an antenna crashes, then 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 different network position than*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. Since requests can happen asynchronously, a relational database denoted *DB* is used to store this data. This choice was made as relational database is able to receive data at any time and let other pro-

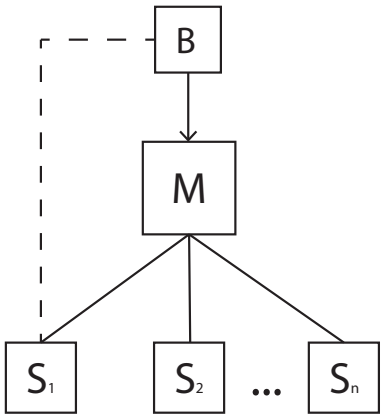

Figure 12: Slave structure.

cesses handle it later.

The response from *M* to *B* 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* was 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 *D* are processes, seen in Figure 13. This improves resource management. The resource management could be to constrain processing power for each process or to distribute the processes on individual machines.

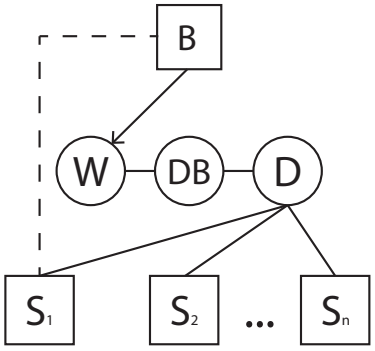

Figure 13: Daemon structure.

3.2 FUNCTIONALITY DISTRIBUTION

The functionality of the system can be derived from the acceptance tests. This section will cover which 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 *W*. There is only one *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 *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_4. 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_{28} 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 creates the need for communication between the instances. Based on Figure 12, which displays the dependencies between *B*, *M*, and *S* this leads to the diagram shown in Figure 14.

The arrows represent a connection, each connection has 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. HTTP requests enable the user to view the web application through *B*, and send information back to the web application. In order to fulfil 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 the content back to *B* to be displayed for the user.

The connection *d* handles the signal described in Section 3.2. This signal covers the initial communication between *S* and *M*. *M* verifies

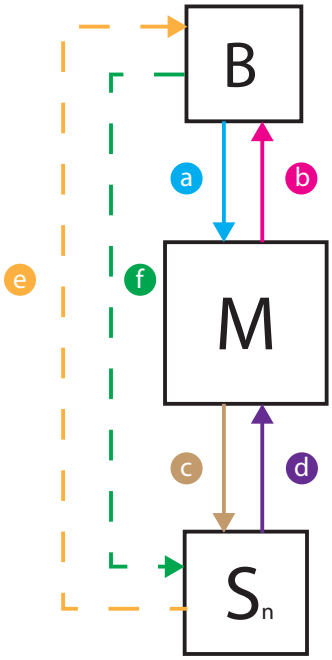

Figure 14: Dataflow diagram of LONE.

the identity of S, by having S send 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 to a customer and ensured that it does not match any existing identifiers. All unique identifiers of each S are recognizable by M. This allows M to determine the identity of each incoming signal. When M receives a signal, and verify the source of the signal as a 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 any location, it can create the problem of S not knowing the identity of B AT_{28} sets the requirement of differentiating between incoming connections to S. If the communication described in connection e and f were running through M, this would be no problem as security would be handled by the already existing sessions on M. The commucation through connections e and f covers a video stream of the drone's video camera and commands for the drone, which we deem to be data intensive. This would increase the load on M, which is not desired since M is assumed to have a high load already. However, since the connections e and f are not running through M, this leaves the problem that S should be able to differentiate between Bs and which to listen to.

The solution chosen to solve this problem is to make *S* create a randomly generated key, denoted session key, that is unique relative to *S* and is 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*, since *M* is able to verify the identity of *B* through the session.

M requests session keys through connection *c*. Since *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.

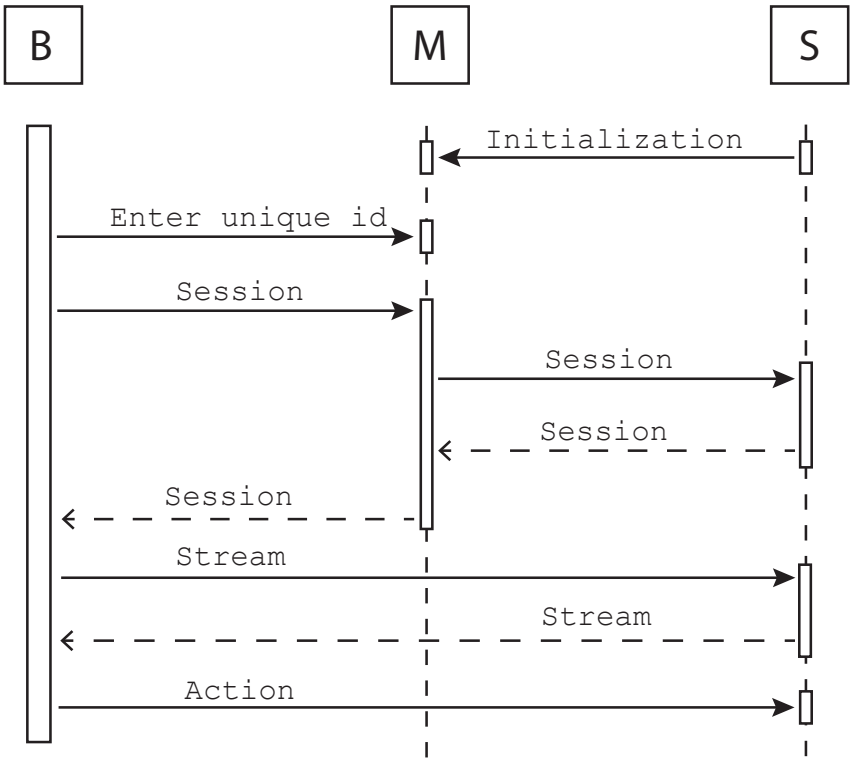

Figure 15: Sequence diagram of the communication network between *B*, *M*, and *S*.

The functionalities of the described connections, can only happen in 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 message 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 I is the amount of interactions required by the user. The amount of interactions should be as low as possible

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

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, has to be assigned to a 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. The users are, however, still categorized as system administrators, as the privileges of system administrator role 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.

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

It is assumed there exists a function $p(u)$, 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 function $p_m(u)$ 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.

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

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:

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

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

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 by several different software companies in cooperation [21].

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 → session_key_tasks.

The relationship between two objects *a* to *b* can be either implicit or explicit. Implicit relationships are illustrated as line from an *a* to *b*, where *b* has a FK from *a* Explicit relationships can be either simple or rich. Simple relationship models are models containing a FK from both *a* and *b* that is connected with a line to both. They are represented in the model diagram with both objects in lowercase and pluralized, e.g. roles_users. Rich relationship models are similar to simple, but has 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 the system and its functionality is restricted. This restriction is based on the identity of the user in the system. An instance of the *User* object represents a user of the system. The *User* object has the attributes `email` and `password`, as seen in Figure 42 in Appendix E.1,

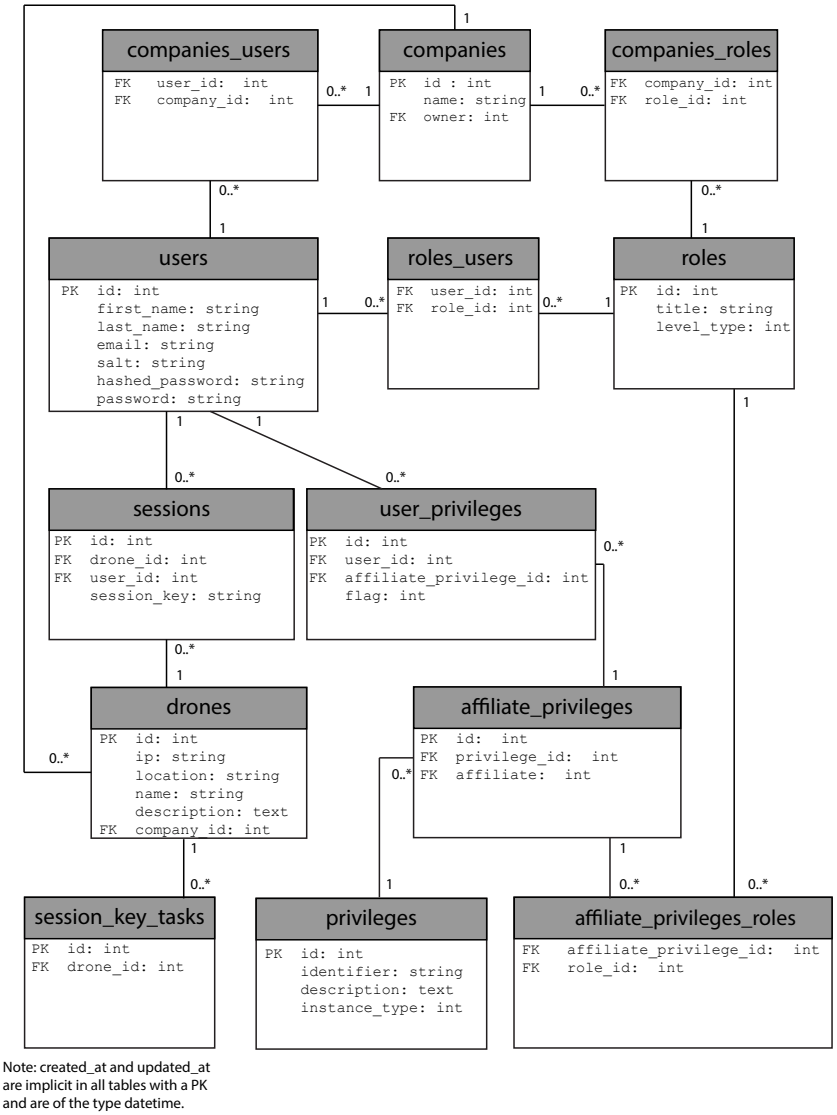

Figure 16: UML Class Diagram of LONE.

these combined form the login credentials needed for a user to authenticate his identity towards the system.

The `email` is publicly available. Therefore the `password` must be protected in order for users to protect their user 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. Storing passwords of users unencrypted would allow a third party to login into the system should he gain access to the database. When a salt is concatenated before hashing it makes it harder for the third party to gain the original passwords, e.g. through a rainbow table [7].

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 43 of Appendix E.1. The physical drone has 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, as described in Section 2.2, therefore there is a need for grouping users by a company. The *Company* object has an `owner`, which is a reference to a *User* object. This is required for the system to know which user has access to all *Affiliate Privilege* objects of the company.

An *Affiliate Privilege* object references a *Privilege*, and combined they form a unique key. This key is associated with a specific functionality in the system. A user is granted access to the functionality by having a relation with the *Affiliate Privilege* that is part of the key. *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 of a privilege gives the right to grant or revoke it from users. *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 control 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. creating 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.

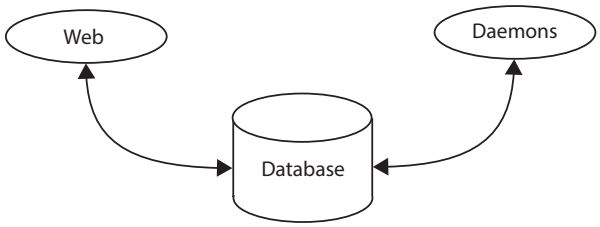

Figure 17: Internal structure of Master.

The web process, *W* will be using the Model-View-Controller (MVC) design pattern [30]. It consists of three layers, model, view, and controller. The concept of MVC is decoupling the different layers to make it possible to develop them with some degree of independence. 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 the code for the controller layer out of the view layer. 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 corresponding 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) [8] 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 asynchronously relative to *W* on *M*. One task will handle the session communication with *S* as described in Section 3.3. The other task receives initialization messages from *S*.

Session communication refers to the session key required to communicate with a drone. *D* will handle session communication with

S by continuously checking *DB* for new session requests. Session requests are inserted in *DB* by *W* when a request is received from *B*, as illustrated in Figure 18. The session communication is handled through the *DB* to account for the asynchronous behavior of the web [36]. 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 *B*. *D* should handle session requests sequentially, 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* receives an initialization message from *S* it will create a drone object and store it in *DB*. In the drone object the `ip`,`location`, and `name` will be retrieved from the initialization message sent from *S*. The `location` is retrieved from the IP and the `name` is the drone's serial. If *M* receives an initialization message from a *S* with a changed IP, it will update the `ip` and `location` of the drone in *DB*. The drone's serial will be described in Section 3.7.

D will handle the session requests as a First In, First Out (FIFO) queue. 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 3.2 and Section 3.3.

1. Establish a wireless connection to the drone.
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), since all the communication with the

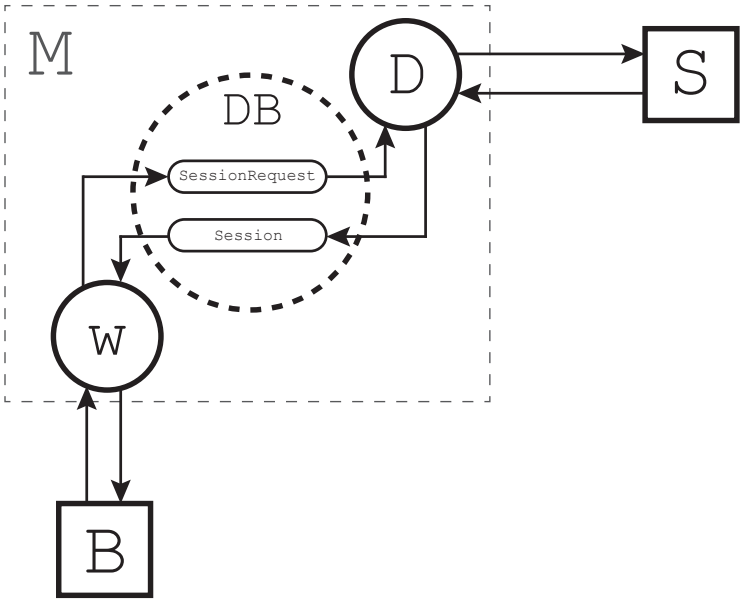

Figure 18: Session key communication.

drone is wireless as described in Appendix B. *S* will automatically attempt to establish a connection to the drone's wireless network when powered on. Following this the initialization message is sent to *M*. When the initialization is successfully completed, *S* is ready to use. Each *S* will have a unique identifier called serial.

The initialization message is sent from *S* to *M*. It will contain *S'* serial. The initialization message is send to *M* when *S* has completed booting.

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*. The session key is critical to keep secure, since it represents the identity of a user, which is required in order to set the right permissions, as stated in Section 2.1. Therefore it is important that the string length is sufficient, since this is stronger than the string complexity [25]. Thus it was decided that a length of 40 characters would be sufficient for the session keys. Following the generation of a session key it is stored on *S* and sent to *M*. While a session key is active on *S* only messages containing it are forwarded to *S'* associated drone. Session keys will be deleted from *S* if the connection times out.

S receives commands from *B* that *S* forwards to the drone, see item 4. The commands are received and converted to a format interpretable by the drone and sent to the drone, see Appendix B.

Item 5 will be explained in Section 3.10, as already existing technologies will be used.

3.8 BROWSER

Users interact with the system by connecting to *M* through *B*. 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 stream, 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 the user to interact with the system he is required to authenticate in order to establish a connection with *W*. A session that holds the identity of *B* is stored on *M*, and it is used to uphold the user's 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. *B* must be able to display the drone's video stream through a multimedia object. To control the drone *B* must request a session key from *M* as illustrated in Figure 15. When controlling the drone the video stream must be available to *B*.

The user interacts with LONE through *B*. AT_1 and AT_2 require 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 33 in Appendix D. The user enters his login credentials, as described in Section 3.5, 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 user's privileges are retrieved through the user 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 unlink a drone. Drones are added automatically to the system through the initialization messages sent by *S*. To make a drone usable in LONE, it must be added to a company as reflected in the object model, as described Section 3.5.

3.9 USER INTERFACE

This section covers the principles and guidelines used in order to create the Graphical User Interface (GUI) of LONE. The layout for all views, as seen in Figure 19, that require authorization to use was created with the use of the Gestalt principles [37]. This layout is a GUI, 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 19. *Content* of the layout is a dynamic placeholder for individual content of each view, while the menu remains the same.

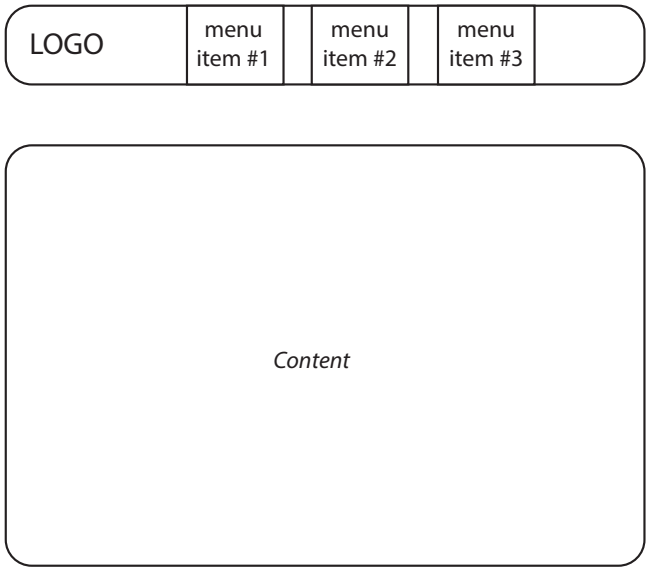

Figure 19: GUI layout.

Since the HTTP protocol is stateless and requires all parameters to be received through requests [13], 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.

3.10 TECHNOLOGIES

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

3.10.1 *Server Operating System*

In this project two operating system families will be considered for use. Each family consists 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

Linux Debian was chosen as the server operating system due to the following reasons:

- Linux Debian is open-source [11], thus it is not necessary to purchase licenses.
- 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 [35].

The operating system chosen will be used for *M* and *S* in LONE.

3.10.2 *Programming Language*

Using Linux Debian as operating system on *M* and *S*, limits the possible options for programming languages, since the chosen programming language must be supported by Linux Debian.

The programming language must have the following:

- Support for communicating with databases, as described Section 3.1.
- Support for network communication, as described Section 3.3.
- Support for object oriented programming, as described Section 3.5.

The considered programming languages are Perl, PHP, Python, and Ruby.

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)* provides a framework for web-development with Ruby.

Ruby is an object-oriented scripting language.

Rails is an open-source web framework for Ruby [20]. Rails provide a set of built-in features, such as native support for the MVC model and Active Records [16]. Rails also contains features, which simplifies configuration management. An example is Migrations. Migrations are a 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 a change to the database with a migration, he can share it with the other developers in his team. The other developers can then update their databases to reflect these changes by running his migration. Additionally migrations makes deployment of systems easier, as the database is automatically setup by running all migrations.

3.10.3 *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 stream and handle some of the computation to reduce the load on *M*. Reducing the load on *M* can be achieved using JavaScript [9].

Another tool that helps reduce computation on *M* is: Asynchronous JavaScript and XML (AJAX). AJAX allows *M* to only compute the needed

pieces of information instead recomputing and resending the entire view. Then JavaScript on *B* uses the information to update the view locally. This also provides the asynchronous processing described in Section 3.8. This approach has several advantages:

- The content of a view can be updated without recomputing the entire page. This is normally not possible with web-development, as the web is stateless [36].
- Using AJAX can improve usability of a web application [6].

These tools ensure that *B* of LONE is be platform independent while providing a better user-experience for the users [6].

As described in Section 3.8 a technology must be used in *B* to display the drone's video stream. 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 [12]. 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 *B*. 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 stream and control it.

3.10.4 *Streaming Technologies*

Developing a video streaming tool is outside the scope of LONE as described in Section 3.7. Video streaming is therefore done using existing streaming technologies.

The video stream sent by the drone to *S* is forwarded to *B* as seen in Figure 15. The video stream is displayed in a Flash application, as described in Section 3.10.3. Flash only natively supports video streams send via Real Time Messaging Protocol (RTMP) [5]. The drone's video stream is H.264 encoded and sent over TCP, as described in Appendix B. Flash is compatible with H.264 encoded video [4]. The drones video stream is however encoded with the PaVE headers as Frame Header, see Appendix B. Flash is incapable of reading PaVE headers. Therefore they must be removed before Flash is

capable of displaying the video stream. To achieve this each *S* must contain functionality capable of reading and encoding the drone's 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 drone's video stream and broadcast it over RTMP without the PaVE headers. The other is using two distinct technologies, with one reading the drone's video stream and forwarding it to a streaming server. The streaming server reads the incoming video stream 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" [2]. It contains a multimedia streaming server called FFserver. FFserver is used for live broadcasts and is capable of decoding and encoding video and audio. It does not contain functionality to decode the PaVE headers, but can output over RTMP. Therefore FFmpeg cannot be used to read the drone's video stream, 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 stream is processed by. As an example a video stream might be decoded and then encoded in a new format. GStreamer does not have an RTMP server. It is however capable of sending a video stream to an RTMP server. GStreamer does not contain functionality to parse PaVE headers. There does however exist an externally developed plugin for GStreamer [24], which contains the functionality to parse PaVE headers. The plugin is named GStreamer PaVE parser (paveparse).

CRTMP is a streaming server capable of streaming to and from a Flash application using the FLV format. It can receive a local stream sent 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 [3], therefore CRTMP will be used as the RTMP server in LONE.

4 IMPLEMENTATION

In this chapter the implementation of LONE's functionality is explained. This chapter only covers selected parts of the implementation. It can be expected, that the implementation of all of the functionality not described in this chapter, has been implemented as 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. Both of these tools will be set up on *S*. Setting up GStreamer and CRTMP enables that the drone's video stream can be taken as input and displayed in a Flash application. This is not possible, however, due to the PaVE headers.

4.1.1 *GStreamer*

As described in Section 3.10.4 the tool GStreamer is used for reading the drone's video stream and forwarding it to CRTMP. GStreamer is a pipeline based tool that has a command line version which can be run using the `gst-launch-0.10` command. 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 20.

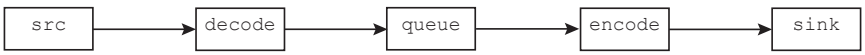

Figure 20: Illustration of a sample pipeline.

The source element is capable of receiving a video stream over a specific protocol from a defined source. As described in Appendix B the drone outputs its video stream using TCP/IP 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 2.

The data available to the next plugin is H.264 encoded video with PaVE headers. The PaVE headers are parsed using the paveparse plugin [24], as described in Section 3.10.4. Paveparse removes the PaVE headers and sends the remaining H.264 data down the pipeline. Fur-

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

Listing 2: Partial GStreamer pipeline illustrating tcpclientsrc.

thermore it ignores lost frames and discards frames sent out of order to reduce delay on the stream. Since the drone streams over TCP/IP and thus no frames are missed, this is necessary to remove potential delay.

As specified in Section 3.10 the video output of GStreamer should be in FLV so that the Flash application on *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 [22]. 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. Since the video stream 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 3:

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

Listing 3: Partial GStreamer pipeline illustrating rtmpsink.

The pipeline that forwards the video stream from the drone can be seen in Listing 4.

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

Listing 4: 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 [18]. The `queue` element is added between every plugin except `tcpclientsrc` and `paveparse`.

The complete pipeline can be seen in Listing 5:

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

Listing 5: Complete GStreamer Pipeline.

4.1.2 *C++ RTMP Server*

As described in 3.10.4 CRTMP is used as the server tool between GStreamer and the Flash application on *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 6.

```
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 6: Snippet of CRTMP flvplayback.lua Configuration.

`Acceptors` define inbound connections to CRTMP, meaning which ports users can connect to. `externalStreams` define outbound connections to CRTMP, meaning the source of the external input. The `externalSteam` seen in Listing 6 takes a video file from an external server as input, and gives the 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, since 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 7, where `server address` is the address of the CRTMP server, `application` is the application where CRTMP does a lookup e.g. a live-stream or local media, and `streamName` is the name of the stream.

When a user connects to CRTMP and the `application` is live, CRTMP will try to find a live stream that corresponds to the link name. If no

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

Listing 7: CRTMP URL Format.

such live stream 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 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 [15, 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 by Flash. 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 on how to interpret it. As a result of this the Flash application and VLC are unable to interpret the video stream. An illustration of the PaVE header problem can be seen in Figure 21.

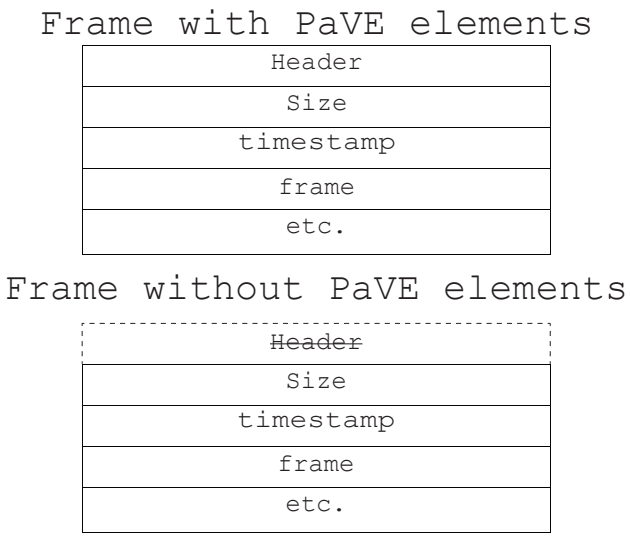

Figure 21: Illustration of video frames with and without PaVe headers.

The PaVE headers result in a situation that makes displaying the video stream in Flash impossible, as the video stream cannot be interpreted by a Flash application neither with nor without them.

4.1.4 *Testing with a Test-input*

The designed solution would work if the drone's video stream used standard headers. The issue is illustrated in Figure 22. 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 in Case 3 in Figure 22 the setup can forward a stream readable by Flash. This can be documented by using a different video source as illustrated by Case 4. For this purpose the GStreamer plugin `videotestsrc` is used. It creates a test video stream as seen in Figure 23, consisting of raw video data.

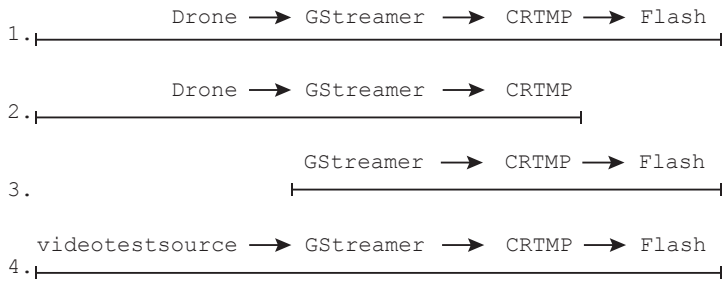

Figure 22: Streaming connections.

A `testvideosrc` pipeline can be seen in Listing 8.

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

Listing 8: GStreamer Pipeline using videotestsrc.

If this test-source is used instead of the drone's video stream, it is possible to display the video stream sent by GStreamer.

This solution can be seen in Figure 24.

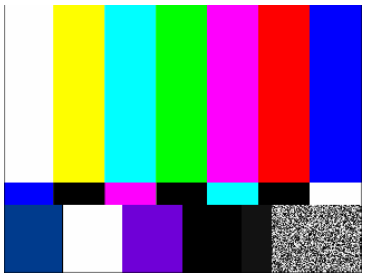

Figure 23: The test video source played with a Flash application.

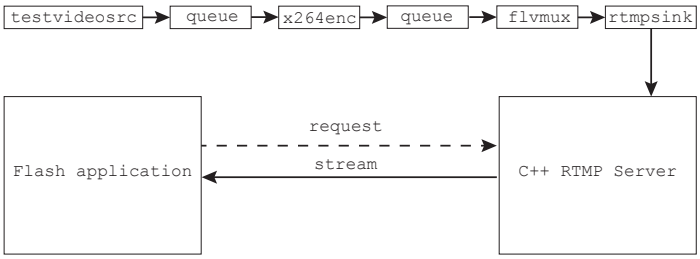

Figure 24: Illustration of the working solution with use of the video test source.

4.1.5 *Implemented Streaming Solution*

The streaming setup used in LONE uses GStreamer. One pipeline that reads the drone's video stream and acts as a server, denoted *SP*, and a client pipeline, denoted *CP*, for displaying the video stream on the client side. The setup can be seen in Figure 25.

Figure 25: Illustration of the implemented solution with xvimagesink.

SP reads the drone's video stream and multicasts it using Real-time Transport Protocol (RTP). RTP is a standardized protocol for sending audio and video packages over an IP network. RTMP is not used, since it is a protocol made specifically for Flash. RTP is better choice for a solution using only GStreamer, 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 plugins `tcpclientsrc` and `paveparse` described in Section 4.1.1.

Video data sent via RTP must be pay-loaded, since a 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 9:

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

Listing 9: 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` is used to describe metadata about the incoming video stream and contains information such as encoding, the protocol it is being sent using, Framerate, etc. The caps are generated by *SP*. The `udpsrc` element of *CP* can be seen in Listing 10.

```
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 10: 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 and decoded after the transfer.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 false. The complete *CP* can be seen in Listing 11.

```
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 11: Client pipeline.

4.2 WEB-INTERFACE

The web-interface is implemented using Rails as described in Section 3.10.2. This framework enforces a number of standards that must be followed. The most significant is CRUD, mentioned in Section 3.6. The web-interface is implemented using the design described in Section 3.6 and Appendix D.

A number of Rails-specific libraries, named *Gems* [10], are used in LONE. Gems are written and maintained by the Rails community. The gems used in LONE are: *mysql2*, *daemons-rails*, *eventmachine*, *sht_rails*, and *swfobject-rails*.

In LONE the relational database MySQL database to store information in. Ruby and Rails is not natively able to connect to a MySQL database. *mysql2* provides an interface for Rails which is used for connecting with the MySQL database [29].

The *daemons-rails* [19] is used for the daemon running on *M*. Ruby has native support for daemons. *daemons-rails* was chosen as opposed to these, as *daemons-rails* can use the model in the rails app and communicate with the database. This used to create *drone* objects whenever a *S* sends an initialization message to *M*.

The *eventmachine* Gem [17] provides an abstraction for communication over TCP/IP. *eventmachine* is used to handle the communication between *M* and *S*. This involves both the initialization messages from *S* to *M* and the communication used for session keys.

Shared Handlebars Template for Rails (SHTRails) [39] enables the use of Handlebars[1] templates in Rails. This allows for sharing a HTML template between Rails and Javascript.

The *SWFObject* Gem [26] is used to display a swf flash object in the view of LONE. This is needed to implement the Flash application needed to display the drone's video feed and to send control commands to it.

5 TESTING

This chapter will cover the testing of LONE, how it was designed and performed. LONE was subject 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 [31].

- Black-box testing
  - Static – Inspection and review of specifications
  - 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[31]. Following the completion of the development the system was subjected 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.

To move LONE to a deployment state more extensive testing is needed. Unit-testing should be applied to ensure the functionality in LONE is fit to use, as well as tests with potential end-users. The end-user tests should primarily involve usability testing, to ensure LONE in intuitive to users. 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.

The dynamic Black-box testing is performed using the acceptance tests 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.

5.2 TEST REPORT

All tests result can be seen in Appendix E. The functionality associated with the tests that passes is fully implemented in LONE. The tests 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 26: Acceptance tests results.

From the acceptance tests results the following can be derived:

- The primary functions of LONE are implemented and working.
- Some of the functionality can be improved and made more user-friendly.
- The failed acceptance tests listed in Figure 26 are not needed for a proof of concept solution. 20 of 28 performed tests passed, and as none of the failed tests were associated with functionality needed for a proof of concept, the result was deemed acceptable.

6 EPILOGUE

LONE is the result of an attempt to apply drone technology in a scalable web based video surveillance system. In this chapter the work and results of LONE is discussed and reflected upon. The conclusion of LONE is presented with the preceding problem statement

6.1 DISCUSSION

A series of design choices were made throughout the development phase of LONE. Some of these will be discussed in this section.

One decision was made to use the programming language Ruby and the framework Ruby on Rails (Rails). Rails contains functionality, which made the development of LONE easier, such as support for the Model-View-Controller (MVC) design pattern and Active Records. This functionality optimized the time spend on development. This is a student project, which means no personal resources are involved, except already preallocated time. This enables us to focus on learning new languages with less concerns about meeting the deadline with the granted resources. This meant that some of our time had to be dedicated to learning these new technologies. Learning a new language involves a trade-off of what the new language provides, in terms of time saved, opposed to the time spent learning it. Had we used technologies in which the whole group had experience, we may have been able to implement the project faster. Such a technology could be PHP, which the whole group has experience with. We decided that the learning process would be worth more than the time we might have saved.

There are some unresolved issues in LONE. These issues can be prioritized according to their need to be resolved in order to have deployable solution.

Whenever a user (Browser B) is trying to access a drone, it must receive a session key from the web application on Master M generated on the Slave S associated with the drone. This key is transferred unencrypted over regular HTTP. This means that the key can be read by viewing packages sent on the network, since it is sent as raw data. This has no impact on the functionality of the system, but it is a security concern that should be addressed before LONE is ready to be deployed.

Another potential security issue is the registration of new *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*. These security issues does not affect the functionality of LONE. These must be resolved before LONE can be deployed.

Some of our time was spent on streaming the video stream from the drone to *B*. The amount of time was more than expected. GStreamer was used and set up with appropriate options on *S* to fetch the stream from the drone. It was not possible to display the video stream in the Flash application on B due to issues with the PaVE headers as described in more detail in Section 4.1.3. Some of our time was spent on this, as displaying the video stream from the drone is one of the important features of LONE. This amount of time was more than expected. Instead we implemented an alternative solution, which did not meet the requirements of LONE. This solution is described in more detail in Section 4.1.5. It requires the user to run a separate application to view the video stream, and have the Flash application focused in the browser at the same time to be able to send commands to the drone.

This solution is fulfilling as a proof of concept. It is possible to both view the video stream from the drone and send commands to it. If LONE was supposed to be deployed, a solution where the video stream is presented in the browser, in the same Flash application that sends the commands, is a requirement.

The time used for implementing the design exceeded our expectations, and was due implementation issues of the streaming. However, since the implementation does not include the design of streaming. This is an issue that has to be resolved before LONE is able to be deployed.

During the design and implementation of LONE we used worksheets to document our thoughts and work. This enabled us to write down design decisions throughout the development process. This allowed us to remember the design decisions during the writing of the report.

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. Based on the use cases a set of acceptance tests were made from which the functionality of LONE was derived. The design was based on this functionality.

LONE was both developed using an iterative process, where a combination of SCRUM and XP was applied. Each iteration contained planning, design, implementation, and testing. This report documents the final results of the design and implementation.

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 developed to support a growing amount of users.

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 drone's video stream, 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 implemented privilege concept. 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. These tests were the base of testing LONE. Only acceptance tests were performed on LONE. These were run to ensure that all of the required functionality was present and working. They did, however, show that some of the functionality defined by the acceptance tests, 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.

LONE in its current state is not ready to deploy. 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

A number of the acceptance tests failed. The tests which were not passed must be implemented and bugs must be corrected before LONE is ready for deployment

Dynamic black-box testing, through acceptance tests, was the only testing performed after development ended on LONE. To move from the current proof of concept state to on ready for deployment, LONE must be subjected to more extensive testing. This testing should include both unit testing to eliminate bugs, and usability testing.

The streaming solution made for LONE is not ideal from an use case perspective. To implement a solution usable in LONE would require developing a streaming tool, capable of parsing PaVE headers and forward a video stream displayable in a Flash application.

Those are the issues that must be resolved in the project's current state There are however a set of additional items that must be addressed to make LONE ready for deployment:

- Radio-controlled drones – It is not an optimal solution that the drones broadcast their own WIFI network that *S* must connect to in order to enable communication with the drone. An alternative solution is communicate with the drone through radio signals, which gives an advantage in terms operational range of

the drone. The maximum range of the Ar Drone's Wi-Fi signal is 50 meters [14], where as a radio controlled drones can have operational lengths of up to 200 kilometers [32].

- Better hardware – The current drones has limited fly time due to the batteries. The Wi-Fi network card used in *S* is of low quality. It has a limited range, and its driver contains a bug which causes *S* to crash whenever the connection to the drone is lost. This hardware needs to be improved before LONE deployed.
- Recording of video stream - The video stream must be recorded on *S* to ensure video documentation is available.

Part I

APPENDIX

A

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 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*.

C

MODEL FUNCITONS

Function	Description
`privileges`	returns all privileges associated with a company

Figure 27: 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 28: Privilege model.

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

Figure 29: Drone model.

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 30: Session model.

Function	Description
`privileges`	Returns the privileges of the role

Figure 31: 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 32: User model.

D

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 33: Access controller.

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

Figure 34: 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 35: 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 36: Drone controller.

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

Figure 37: 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 38: 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 39: 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 40: 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 41: Acceptance tests 2.

F

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 42: 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 43: 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 44: 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 45: 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 46: 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 47: 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 48: Session object.

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

Figure 49: 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 50: User Privileges relationships.

BIBLIOGRAPHY

[1] URL http://handlebarsjs.com.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] Jeff Atwood. Rainbow hash cracking. *Coding Horror*, September 2007. URL http://www.codinghorror.com/blog/2007/09/rainbow-hash-cracking.html.

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

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

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

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

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

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

[14] Craig Dunning. See through the eyes of your own flying machine - parrot ar.drone 2.0 review. *The Daily Telegraph*, April 2012. URL http://www.dailytelegraph.com.au/lifestyle/parrot-second-generation-ardrone-20-wi-fi-quadricopter/story-e6frf00i-1226333390462.

[15] 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.

[16] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison–Wesley, 2003.

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

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

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

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

[28] 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.

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

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

[31] Ron Patton. *Software Testing*. Sams Publishing, 2nd, edition, 2006. ISBN 0-672-32798-8.

[32] John Pike. Unmanned aerial vehicles (uavs). URL http://www.globalsecurity.org/intell/systems/uav-intro.htm.

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

[34] Lonni Rasmussen. OvervÃ¥gningens dilemma. *TeknologirÃ¥det*, 2001. URL http://www.tekno.dk/subpage.php3?page=statisk/tema/overvaagning/dilemma.html&toppic=oplysning.

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

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

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

[38] 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.

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

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of December 19, 2012 (`classicthesis` version 4.1).

DECLARATION

Aalborg, December 2012

Anders Eiler

Bjarke Hesthaven Søndergaard

Esben Pilgaard Møller

Rasmus Steiniche

Thomas Kobber Panum