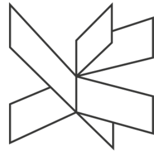


CLOUD COMPUTING FOR END USERS

PROJECT REPORT

Kenneth Nørholm 254309
Krystof Spiller 253812

supervised by
Lars Bech Sørensen

74442 characters (not including spaces)

Software Engineering, 7th semester
December 16, 2019

Document versions:

Version	Change	Date
0.1.0	Initial structure following ICT specific guidelines	2019/08/19
0.2.0	Start of versioning	2019/10/16
0.3.0	Use case descriptions reorganized and expanded; Analysis section reviewed; new Middleware section in Implementation; new File servermodule section in Backend design	2019/11/04
0.4.0	Design - added file capabilities; Implementation - revised Middleware section; Analysis - migrated GUI design from Design to Analysis; Test - started description of need to have parts of the use cases	2019/11/18
0.5.0	Design - described events capture in frontend and keyboard control API in backend; Test - all use cases described and moved to appendix, left only illustrative one; Discussion - started; added non-functional requirement; figures updated to match current design	2019/11/25
0.6.0	Added Abstract and User manual (appendix); Test, Results and discussion, Conclusion and Project future considered release candidates; minor additions to Use case descriptions and Delimitations; added figure 13 to Middleware design	2019/12/02
1.0.0	Minor changes after going through finalization checklist; changes to Glossary; minor figure updates; created a source code appendix	2019/12/16

Contents

Abstract 6

Glossary 7

1 Introduction 9

2 Analysis 11

2.1 Requirements 11

2.1.1 User stories 11
2.1.2 Non-functional requirements 11
2.1.3 Use cases 11
2.2 Use case descriptions 12
2.2.1 Use case description: Manage account 13
2.2.2 Use case description: Launch a specific application 14
2.2.3 Use case description: Control a running application 14
2.2.4 Use case description: Manage personal files in the system 16
2.3 Look and feel of GUI 18
2.3.1 Main window 18
2.3.2 Login and create account 18
2.3.3 Applications view 20
2.3.4 Files view 21
2.3.5 Slave application window 22
2.3.6 Window controls 22
2.4 Delimitations 23
2.5 Domain model 23
2.6 Chosen technologies 24

3 Design **25**
3.1 Overall system design 25
3.2 Frontend design 26
3.2.1 Electron 27
3.2.2 Communication module 28
3.2.3 Event capture 28
3.2.4 Web technologies 28
3.3 Middleware design 29
3.3.1 NetMQ / ØMQ 30
3.3.2 Generic communication library 30
3.3.3 Usage in distributed system 35
3.4 Backend design 36
3.4.1 Backend design overview 37
3.4.2 Docker 37
3.4.3 Server module 38
3.4.4 Slave-owner servermodule 38
3.4.5 File servermodule 38
3.4.6 Database servermodule 39
3.4.7 Slave controller 40

4 Implementation **44**
4.1 Frontend 44
4.1.1 Image receiver 44
4.1.2 React 45
4.2 Middleware 46
4.2.1 Encoding 46
4.2.2 Middleware library 47

5 Test **51**
5.1 Test of use cases 51
5.2 Test of non-functional requirements 54
5.2.1 Non-functional requirement 1 (Windows 10) 54
5.2.2 Non-functional requirement 2 (CPU utilization) 54
5.2.3 Non-functional requirement 3 (command delay) 55

6 Results and discussion **56**
6.1 Table of test results 56
6.2 Discussion of test results 57
6.2.1 Discussion of failed non-functional requirement 2 (CPU utilization) 57
6.2.2 Discussion of failed non-functional requirement 3 (command delay) 57
6.3 General discussion 58

7 Conclusion **59**

8 Project future **60**
8.1 Business model 60
8.2 Security and privacy 60
8.3 Availability 61
8.4 Legal matters 61
8.5 Hardware provisioning 61
8.6 Scalability 61
8.7 Application selection 61
8.8 Additional features 62
8.8.1 Automatic slave initialization 62
8.8.2 Store application configuration 62
8.8.3 Automatic application updates 62
8.8.4 Integrated system augmentations 62

References **63**

Appendices **67**

A Test of use cases **67**

B Source code **74**

C Project description **75**

D User manual **85**

E Authorship **86**

List of Figures

1 *Use case* diagram . . . 12
2 Keyboard layout with key groups . . . 15
3 Login GUI within *main window* . . . 19
4 Create account GUI within *main window* . . . 19
5 Applications view and navigation GUI within *main window* . . . 20
6 Files view and navigation GUI within *main window* . . . 21
7 *Slave application window* . . . 22
8 Domain model . . . 24
9 Overall system structure . . . 26
10 *Client application* overall design . . . 27
11 Objects sendable via *generic communication library* . . . 32
12 Reduced class diagram for the *generic communication library* . . . 33
13 Simplified overview of the connections in the distributed system 36
14 Client-to-servermodules communication library inheritance overview . . . 37
15 ER diagram showing the `User` table . . . 40
16 Client-to-slave communication library inheritance overview . . . 41
17 Broken image icon . . . 44

Listings

1 *Slave application window* interval for updating an image . . . 44
2 Selective rendering based on `loggedIn` variable . . . 45
3 Usage of a feature flag . . . 45
4 Sample of the `Encoding` class . . . 46
5 Implementation of proxy method for remote method invocation . 48
6 `WrapCallBack` method . . . 48
7 Method `SendMessage` . . . 49
8 Method `ReceiveSendable` . . . 50

Abstract

This project envisions a system that alleviates several disadvantages that exist in the way computing is done today. A fundamental problem is that capabilities of a computer are given and limited by the components it contains. If a laptop is bought for the purpose of use while traveling and light office work, it cannot be expected that the same computer can run power-hungry applications. All in all, the disadvantages of the status quo can be summarized as limiting, uncomfortable, costly and wasteful.

The envisioned system moves the computation of any application to servers and provides a computationally light application that can be used on the end users' computers to access and interact with the applications that are being run on the servers.

Furthermore, the envisioned system stores users' files on servers where it is ready to be used by the applications running on the servers or updated from the client application running on the end user's computer.

This also means that a user can access the system, with all their configuration, applications and data, from any computer with Internet access just by logging in.

The developed system contains all the core functionality of the envisioned system. That is not to say that the developed system is a minimum viable product, as the system is lacking many critical features. However, if the envisioned system was to be fully implemented, it could significantly improve the way computing is done today.

Used technologies: C# .NET Core 2.2, Electron, React, NetMQ, Python with PyAutoGUI, Hyper-V and Docker.

Source code can be found in appendix B.

Glossary

This glossary lists terms used in this report and specifies how these terms are emphasized from within the text.

In this report text that refers to code is written with a `monospaced` font.

Terminology

Terms listed here are *italicized* in the text. When used, it refers to a particular meaning described in the list below:

- *Server suite* - consists of *servermodules* and *slave modules*
  - *Servermodule* - part of a *server suite*. List of them follows:
    - *Server module*
    - *Slave-owner servermodule*
    - *Database servermodule*
    - *File servermodule*
  - *Slave module* - combined unit of a *slave controller* running on a *slave*
    - *Slave* - virtual machine responsible for running a single application that is streamed to and controlled from the *client application*
    - *Slave controller* - software that runs on the *slave*
- *Client application* - native (Windows) application; made up of *main application window*, *slave application window* and *communication module*
  - *Main application window* or just *main window* - window that appears after launching the *client application* showing GUI for login, creating an account, applications view and files view
  - *Slave application window* - window that displays an application running on *slave module*
  - *Communication module* - mediates communication between *client application* and *server suite*
- *Generic communication library* - custom made middleware library that is used both for the communication between the *client application* and *servermodules* as well as for communication between the *client application* and *slave module*
- *Use case* - collection of *need to have* and *nice to have scenarios* that group meaningfully
  - *Scenario* - part of a *use case* defined by a series of steps that are needed to achieve it; can be either *need to have* or *nice to have*
    - *Need to have* - the *scenarios* of a *use case* that must be included in the project
    - *Nice to have* - the *scenarios* of a *use case* that may be included if time permits

Acronyms, initialisms and abbreviations

Terms listed here are written in ALL CAPS in the text without any other emphasis.

- API - Application Programming Interface
- AWS - Amazon Web Services
- CGI - Common Gateway Interface
- CIA - Confidentiality, Integrity, Availability
- CLI - Command Line Interface
- DHCP - Dynamic Host Configuration Protocol
- DOM - Document Object Model
- ER - Entity-Relationship (model or diagram)
- IPC - Inter-Process Communication
- JSON - JavaScript Object Notation
- TCP - Transmission Control Protocol

1 Introduction

This introduction is based on the project description found in appendix C.

Some of the disadvantages of the status quo in computing for regular users where they need to own the hardware that does the actual computing are examined in the following paragraphs.

First and foremost, the hardware the user bought has only a limited computational potential or use case that stays the same for the rest of the hardware lifetime. This means that if this hardware has been bought for ordinary office work, one cannot expect to be able to run on it power-hungry applications such as Adobe Premiere Pro (Puget Systems, n.d.), Autodesk 3ds Max, Trimble SketchUp and similar, as well. On the other hand, in case of gaming consoles, which is just another piece of computational hardware many people buy (Gilbert, 2018), one cannot expect to be able to do any office work. A lot of hardware is also made for a specific form factor further limiting the machine's potential. Consider for example the constrains imposed on laptop manufacturers.

Second, the hardware needs to be exchanged every so often, on average every 5 years (LaMarco, 2018), for a new one because of the hardware obsolescence and therefore lacking computational performance. This necessarily requires some time to be spent selecting the new model and setting it up, as well as paying the upfront cost of the computer. Furthermore, setup of a new computer can be a frustration with installing all of the software from the previous machine and copying the existing data. In case of laptops it means exchanging the whole machine instead of only the parts involved in computation, which is also unnecessarily wasteful.

Third, the hardware is tied to a certain operating system that allows to use features and applications available only on that system. Although it is possible to run many operating systems on one machine, it is nonetheless problematic to run two applications, that are each available only on a different operating system, at the same time.

Fourth, the fact that the user and only the user owns and uses this hardware means that it in fact sits idle and unused most of the time (Alvarez, 2009). This strategy is wasteful, especially if the relative ease of centralizing processing power, which allows for a much higher utilization (Dignan, 2019), is considered. Assume a conservative estimate that the hardware is being used for 25% of the time and stays idle for the remaining 75%. This means that the world needs four times more hardware than if the hardware would be used non-stop without being idle. This project continues the trend of getting "more from less" and "swap(ping) atoms for bits" (McAfee, 2019).

Fifth, in the current paradigm, the user is responsible for updating the applications, which takes extra effort and is therefore a nuisance for the regular users.

Lastly, if a user does not have the hardware with them, they cannot access their machine and use it. Data sharing services such as Dropbox, Google Drive or Microsoft OneDrive (Rouse, n.d.) allow users to put their data into cloud storage,

making them accessible from every computer with an Internet access. As of now, however, there does not exist a solution that would provide the same comfort accessing a users' applications.

This bachelor project looks into a system that alleviates the aforementioned disadvantages. A system that is inspired by the historical approach to computing by mainframe and client (Beach, 2000). This mimics the approach of cloud computing services (GURU99, 2019). Only in this case, it is directed at end users rather than businesses. Specifically, the system explored in this bachelor project is that of running the applications in the cloud and streaming them to the *client application* while allowing the user to use them as usual, similar to how remote desktop works. The main technical challenge in this report can be summarized as "how to use a GUI application from a computer while the application is running in the cloud".

Consider what a system needs to accomplish for the user.

It must allow the user to use any remotely running application as if it would be running locally so that the user is able to work with it as usual. It must also act as a central storage for all of the user's data.

2 Analysis

This section specifies what can be expected from this project.

2.1 Requirements

For this project, the functional requirements are stated in the form of *use cases*. In order to create use cases, some overarching user stories are made. Furthermore, non-functional requirements are stated in a numbered list.

2.1.1 User stories

1. As a user, I want to use any application from a low-end computer.
2. As a user, I want the system to be personalizable.
3. As a user, I want to be able to work with files in the system.

2.1.2 Non-functional requirements

1. The *client application* must run on Windows 10.
2. The *client application* must be able to run on a low-end laptop CPU with an average CPU Mark score of 4967 (PassMark, 2019) with three concurrent *client applications* running, never exceeding 30% utilization.
3. The delay from a mouse or keyboard command is given until the result of execution being shown in the *slave application window* must never exceed 3 seconds.

2.1.3 Use cases

Figure 1 shows the *use case* diagram for the system. This diagram is intended to be used as a basis of discussion to make sure that all of the necessary functionality is covered.

The *use case* diagram displays both *use cases* and actors. The *use cases* represent the contractual commitments. The actors are types of user the functionality is needed for.

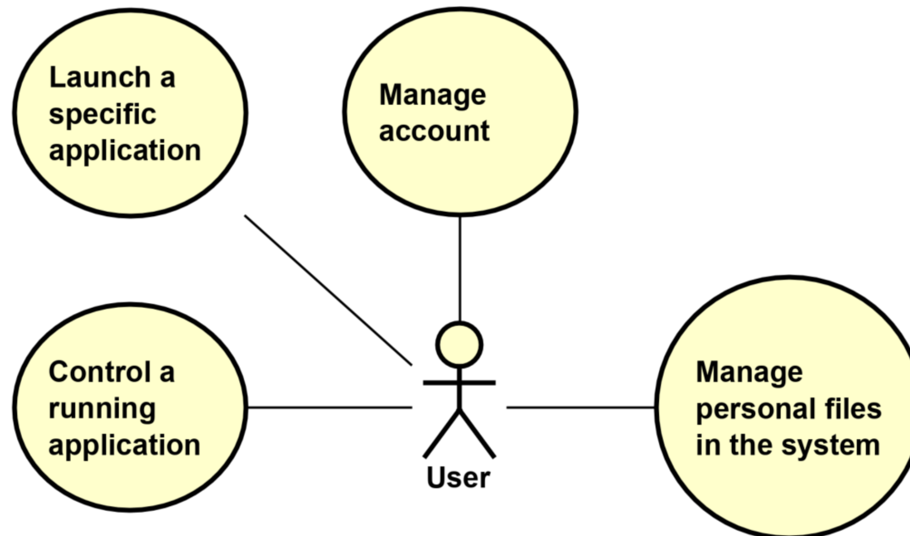


Figure 1: *Use case* diagram

Actor description

- **Actor** : User
  A user is a person who uses the system from the *client application*. A user can have many different motives for using the system. As an example, three different users are considered:
  - A content creator starting out could use this system for resource intensive tasks such as video rendering and editing.
  - A regular computer user who uses their computer for ordinary activities but finds that a computer is too expensive and would like to have a much cheaper option, even if this would require an Internet connection to function.
  - An advanced user that is able to utilize other advantages of the system, such as being able to access their environment from any computer with Internet connection or using any application notwithstanding the operating system on which it runs.

2.2 Use case descriptions

This section elaborates the *use cases* in more detail, specifying which parts of a *use case* are *need to have* and *nice to have*. Only the *scenarios* deemed as *need to have* are further elaborated.

2.2.1 Use case description: Manage account

Actors: User
Elaboration: This *use case* involves creating an account, logging in to an account that is already created, logging out of an account that is logged in, updating an account's information and deleting an account.

Need to have:

1. Create account
2. Login to account
3. Logout of account

Nice to have:

1. Update account information
2. Delete account

Scenario 1 - Create account
Precondition: Having launched the *client application*
Post-condition: Account has been created and user is logged in
Scenario steps:

1. Press "Create account"
2. Enter valid required information
3. Press "Create account and login"

Scenario 2 - Login to account
Precondition: Having launched the *client application* and already having created an account
Post-condition: User is logged in
Scenario steps:

1. Enter required information
2. Press "Login"

Scenario 3 - Logout of account
Precondition: Having launched the *client application* and be logged into an account
Post-condition: The login form is shown
Scenario steps:

1. Click on settings menu
2. Click on "Logout" from the context menu

2.2.2 Use case description: Launch a specific application

Actors: User
Elaboration: This *use case* encompasses being able to launch an application from a *client application* and streaming the application running on the *slave module* to a *slave application window*.

The streaming of the application comprises of multiple media streams coming both from and to the *slave module*. The most important is streaming the visual representation of the application running on the *slave module* in form of a video or images to the *slave application window*. A preferred approach is streaming a video that includes the audio feed as well as justified later in section 3.4.7.

Need to have:

1. Streaming visual representation of the application in any way

Nice to have:

1. Streaming visual representation of the application as a video
2. Streaming audio from the application to the *slave application window*
3. Streaming audio input (e.g. from a microphone) from the *client application* to the *slave module*
4. Streaming video input (e.g. from a webcam) from the *client application* to the *slave module*

Scenario 1 - Launch a specific application
Precondition: Having launched the *client application* and be logged in
Post-condition: New window is created that after initialization shows the selected application
Scenario steps:

1. Navigate to the applications tab
2. Find the application
3. Click the application to launch

2.2.3 Use case description: Control a running application

Actors: User
Elaboration: This *use case* entails remote mouse control and keyboard control of a running application.

Mouse control is an important part of controlling an application, as most applications in use by end users are GUI applications. It can be further broken down into individual parts. There is movement of the mouse itself, its left and

right mouse button as well as the scroll wheel.[1] The mouse buttons produce both down and up events.

The keyboard is another important control device. A keyboard layout that is used as a reference to different key groups can be seen in figure 2. To fully support keyboard control all of the keys should be supported. Key clicks are just like in the case of a mouse made up of both down and up events. These need to be handled separately as it is otherwise not possible to use keyboard shortcuts, which often require several down events before the keys can be released.

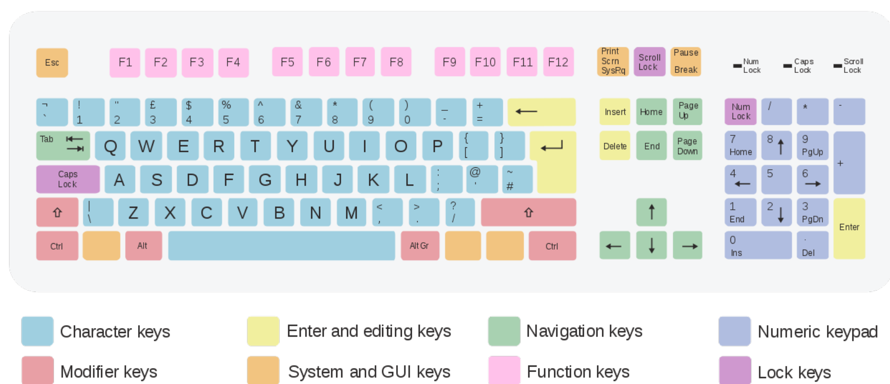


Figure 2: Keyboard layout with key groups (Wikimedia Commons, 2018)

Need to have:

1. Use of left and right mouse buttons, both down and up events
2. Keyboard control - Character keys, enter and backspace, both down and up events

Nice to have:

1. Continuous mouse position update
2. Scrolling
3. All remaining keyboard keys, both down and up events
4. Resize the *slave application window*
5. Changing the local cursor so it matches the one on *slave module*

[1]Other potential mouse actions exists as well, such as back, forward and macro buttons on both mouse and keyboard. For simplicity's sake, these are not considered.

Scenario 1 - Use of left and right mouse buttons, both up and down events
Precondition: Having launched a specific application
Post-condition: See that the mouse events were activated
Scenario steps:

1. Hover the mouse above the *slave application window*
2. Press down on either left or right mouse button
3. Optional: move the mouse
4. Release the mouse button to activate the up event

Scenario 2 - Keyboard control - Character keys, enter and backspace, both down and up events
Precondition: Having launched a specific application and being in a state where typing on the keyboard produces an observable outcome
Post-condition: See that the expected key output occurred
Scenario steps:

1. Press key
2. Release key

2.2.4 Use case description: Manage personal files in the system

Actors: User
Elaboration: This *use case* covers upload of files to the system from the local computer, download of files to the local computer from the system, usage of files with the *slave module*, which includes sending a file from the system to the *slave module*, as well as saving a file from the *slave module* to the system. Supporting these actions does not substitute a regular file explorer and is considered as only a primitive file management.

Need to have:

1. Upload file from local computer to the system
2. Download file to local computer from the system
3. Use file in the system from a *slave module*
4. Get a file from a running application to the system

Nice to have:

1. Rename file in the system
2. Organize files in the system using folders

Scenario 1 - Upload files
Precondition: Having launched the *client application*, be logged in and having navigated to the 'Files' tab
Post-condition: The uploaded file appears in the list of files
Scenario steps:

1. Press "Upload file" button
2. Select a file using the file explorer

Scenario 2 - Download file
Precondition: Already having at least one file in the system
Post-condition: The selected file is downloaded to "Downloads" folder on the local PC
Scenario steps:

1. Select a file
2. Press "Download file" button

Scenario 3 - Use file already in the system from within an application
Precondition: Already having at least one file in the system and having a running application
Post-condition: The selected file can be opened in the application
Scenario steps:

1. Select the file to send
2. From the *main application window*, press "Send file to an application" button.
3. From the dropdown menu select an application to send the selected file to.
4. Open the file from within an application in a usual way. The file is found in the folder "ccfeu-files" located in Desktop.

Scenario 4 - Get a file from a running application to the system
Precondition: Having a running application
Post-condition: The list of files is updated and changes are present
Scenario steps:

1. Save changes to specific folder ("ccfeu-files" located in Desktop)
2. Close the *slave application window*
3. When the *slave application window* is closed, then the files are saved to the system.

2.3 Look and feel of GUI

By analyzing the *use case* description *scenarios*, it was decided to go with an approach where one window (*main window*) is responsible for every user interaction that is not directly interacting with an application running on the *slave module* and all the other windows (*slave application window*) interact with a single application running on the *slave module* and are launched from the *main window*. Furthermore, as the *main window* has to have GUI for login and sign up forms and for lists of applications and files, the preferred window shape is an elongated rectangle.

As the chosen technologies (section 2.6) allow building the GUI as a web application, Bootstrap, a popular CSS framework, can be used. This choice then inspires the GUI designs.

This section shows mockups of the GUI design and describes the thinking behind them. Figures in this section are cropped to save space.

The user manual for the system can be found in appendix D.

2.3.1 Main window

Main window has GUI for the user to login or create an account, applications view and files view. By default it is a rectangular window with an aspect ration of 2:3 so it is higher than it is wide as this shape is better for showing a list of items, such as applications or files.

2.3.2 Login and create account

The login form is shown first when the application is launched and can be seen in figure 3. The create account form is shown in figure 4 and is accessible from the login form by clicking the light gray button in the bottom of the form. Create account form has a similar button to get back to the login form.

Both of the forms require only email address and password, as this is the only information that is used. The blue button submits the form and if the submitted information is incorrect — in the case of a login it is a nonexistent combination of the email address and password and in the case of creating an account it is an email address for which an account already exists — relevant error message is shown above the submit button.

In a production ready system, more information would be required when creating an account. Some additional fields could be a username and a confirm password. Furthermore, when logging in, either email or username could be used. There should also be a checkbox to remember user. Moreover, validation of input and measuring of password strength should be done in the create account form. It should also be verified that the email is valid.

Figure 3: Login GUI within *main window*

Figure 4: Create account GUI within *main window*

2.3.3 Applications view

Figure 5 shows an applications view and navigation GUI within the *main window*. This is the view that appears after a successful login.

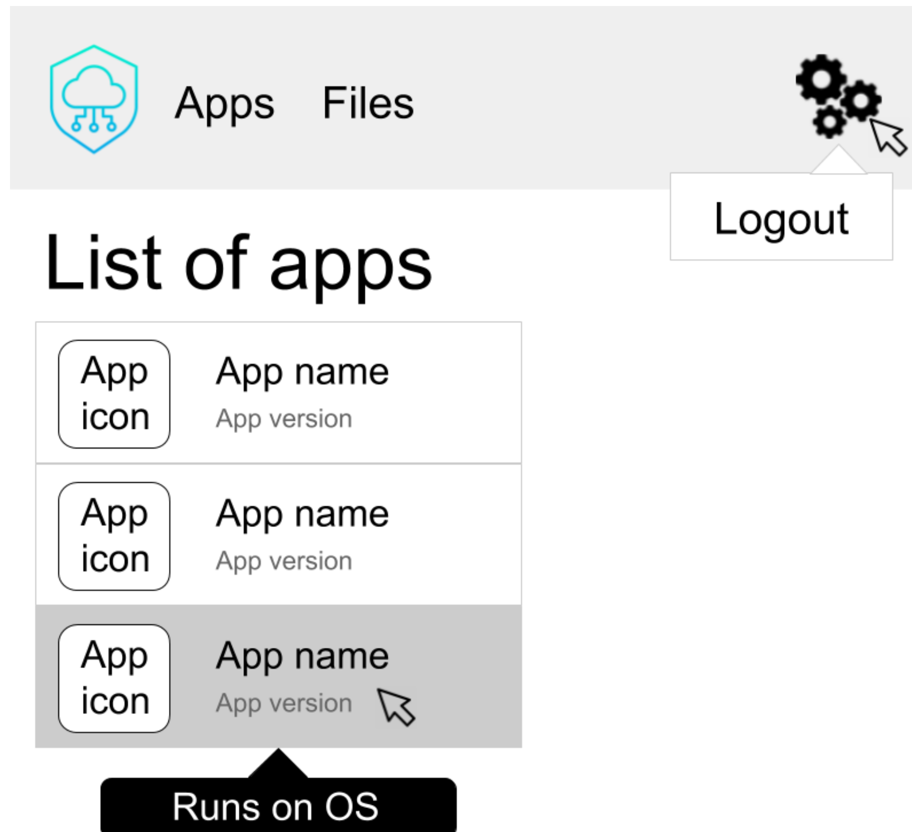


Figure 5: Applications view and navigation GUI within *main window*

It shows a navigation bar in the top having options for either applications view or files view and settings icon to the right. The cursor signifies that there is an additional functionality when the element is being interacted with either by hovering over or clicking on it. By clicking on the settings icon a dropdown menu appears showing an option to logout. The navigation bar is a common element for both the applications view and files view.

Below the navigation bar is a list of applications that can be launched. Each item in the list shows an icon,[2] name and version of the application in question. Additionally, when the item is being hovered over, it shows which operating system it runs on as the same application can be available for many operating

[2] In the current version, only a placeholder icon is shown.

systems and be slightly different either in their look and feel or the offered functionality. Clicking on the item creates a new *slave application window* with the selected application.

2.3.4 Files view

Figure 6 shows a files view and navigation GUI within the *main window*. This is the view that appears after a clicking of "Files" in the navigation bar.

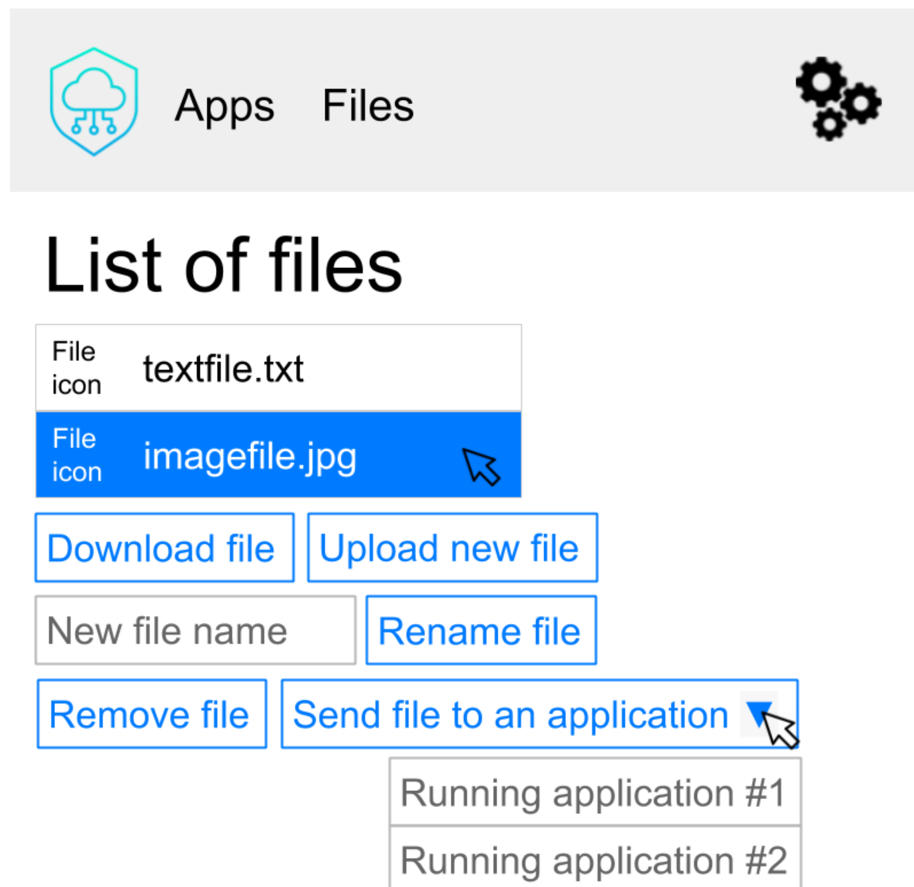


Figure 6: Files view and navigation GUI within *main window*

The files view shows a list of files that are uploaded in the system together with a couple of buttons to interact with the files. Each item in the list shows a file icon[3] and a file name. When the item is clicked, it becomes active and

[3]As for the icons on the applications view, only a placeholder icon is shown.

the background changes to blue. When a file is selected, it can be downloaded, removed, sent to a running application or renamed. To rename a file, new file name needs to be given in the field next to "Rename file" button. Finally, there is also a button to upload a new file to the system, that opens a file dialog and allows the user to select a file to upload.

2.3.5 Slave application window

Figure 7 shows a *slave application window*. The design here is straightforward as it is just an image of the application running on the *slave module*. The size of the *slave application window* is variable and depends on the application that is being run.

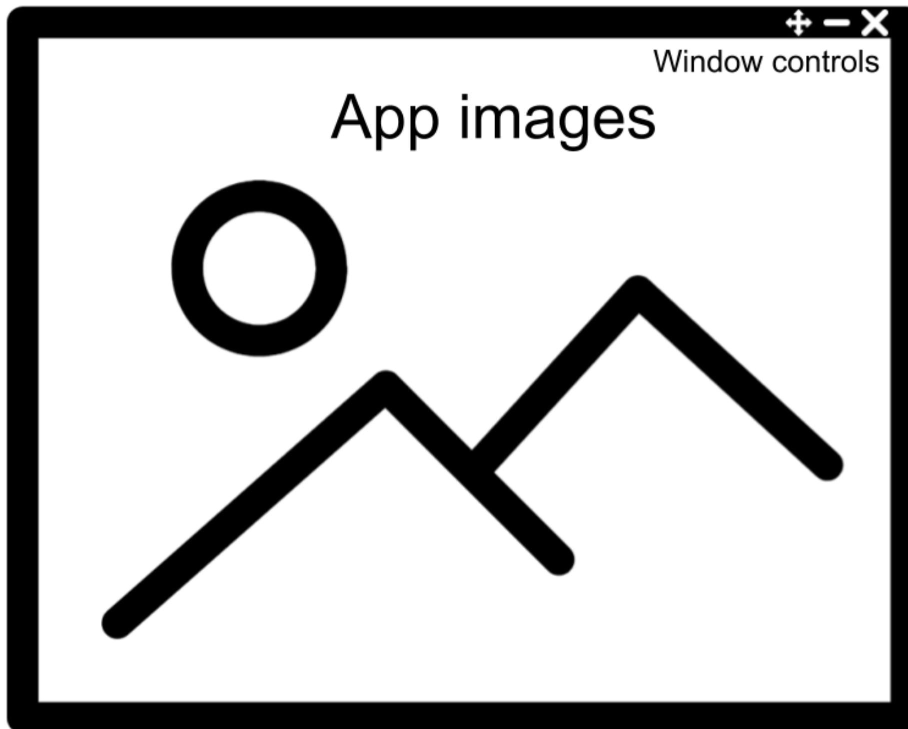


Figure 7: *Slave application window*

2.3.6 Window controls

It is important to note that because both *main window* and *slave application window* are frameless,[4] some elements allowing a basic window control are neces-

[4]Meaning without a chrome as described in Mozilla Contributors (2019a)

sary. Figure 7 is showing elements for closing, minimizing and dragging a window.

2.4 Delimitations

If not stated otherwise, functionality is delimited due to time constraints.

There are *use cases* for an actor called administrator that were delimited. Administrator is responsible for keeping the system running and would therefore be a technically trained person. She would control the system via CLI and she would be able to administer the *server suite*, including user account and application management. These *use cases* are delimited as the need for this functionality only arises in production.

Security, such as encryption of network communication and hashing of passwords, is not considered in this project as it only serves a demonstration purpose.

Management of account data, such as a possibility to change email and password or delete an account directly from the *client application* is also delimited.

The *scenario* for sending a file in the system, to an application described in section 2.2.4 does not take into consideration more than one instance of the same application opened at the same time.

Optimally, it would be possible to organize files into folders. This is to allow users to impose structure on their files and thereby become more productive when working with the system. However, this feature is delimited.

The optimal system would also support streaming audio from the client's microphone as well as video from the client's webcam to the *slave module* in order to increase the number of supported applications, as webcam and microphone are an integral part of some applications, and as such these applications would not be usable without this functionality. However, the specified functionality is not a part of this project.

Another element for the system to be considered production ready is the possibility to resize the *slave application window*. However, as this is a usability feature and not a feature that is necessary to show the feasibility of the system, this is not a part of this project.

2.5 Domain model

A list of domain entities that comprise the domain model are identified from the *use case* descriptions in section 2.2:

1. File
2. Application
3. User Account
4. Client

These entities can then be modeled in a domain model diagram.

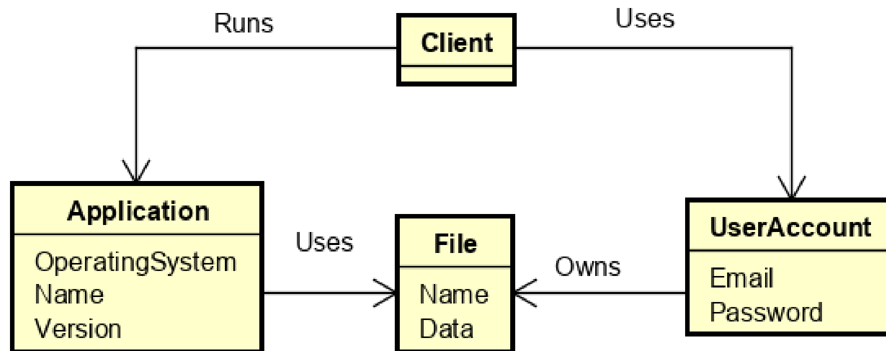


Figure 8: Domain model

Figure 8 shows that the client runs applications,[5] that in turn uses files. Furthermore, the client also makes use of a user account that owns files.

2.6 Chosen technologies

Having completed the analysis of what the solution must be able to do, some technologies have been settled on as the main technologies. These are listed below together with a short description as well as a reason for being chosen.

1. C# .NET Core
   - .NET Core is a general purpose cross-platform programming framework.
   - Chosen because the project team has much experience with .NET and at the same is a great tool for the job at hand.
2. Electron
   - Framework that allows the use of web technologies for development of native cross-platform applications. A longer description of Electron is in section 3.2.1.
   - Chosen because it allows creating cross-platform applications from a single code base as well as a pilot project to try out how Electron works in a project.
3. Docker

[5] Applications running remotely

- System that makes it easy to containerize software to simplify deployment and management of applications. A longer description of Docker can be found in section 3.4.2.
- Chosen both as a pilot project and because of the containerization that Docker facilitates.

4. NetMQ

   - NetMQ is a message queue, network communication library, that supports asynchronous messaging. A longer description of NetMQ can be found in section 3.3.1
   - This technology was chosen because it is made for .NET, as well as professional interest from one of the team members.

3 Design

This section of the report covers the system design, including a short description of specific technologies used to fulfill all the *use cases* and the non-functional requirements. It is separated into four sections.

The first section 3.1 describes the overall system design.

Section 3.2 describes the design of the frontend. That includes both GUI and communication with the backend.

Section 3.3 describes the design of the middleware that is used to facilitate the communication between the nodes.

Section 3.4 describes the design of the backend. That is how the different parts of the backend are designed as to separate out the concerns of the *server suite* into independent units.

3.1 Overall system design

The overall system architecture can be seen in figure 9. The domain entities from figure 8 can be mapped to their own block in the *server suite*.

That is, the user accounts are stored in the database, for which the access is controlled by the *database servermodule*.

The files are stored in the *file servermodule*, which is also responsible for keeping track of file ownership by different accounts and by extension only making those files visible to that account.

The applications are run by a *slave module*. *Slave-owner servermodule* keeps track of *slave modules*, their connection information and which applications and operating system they run.

And finally, the client is represented by the *client application* in figure 9. Here, the *client application* is responsible for communicating with the *servermodules*, and the *slave application windows* are responsible for direct communication with the *slave modules*. It is therefore the responsibility of the *slave application window*

to receive images from the *slave module* and output them to the *client application*. Furthermore, it is also the responsibility of the *slave application windows* to capture events, such as mouse and keyboard events and send those to the *slave module*. As such, each *slave module* has its corresponding *slave application window*.

The communication between the modules happen using the *generic communication library* that is described in section 3.3.2. Due to this fact, the communication module of the *client application* needs to be implemented in C# as the classes needed for invoking remote methods with the *generic communication library* are implemented in C#.

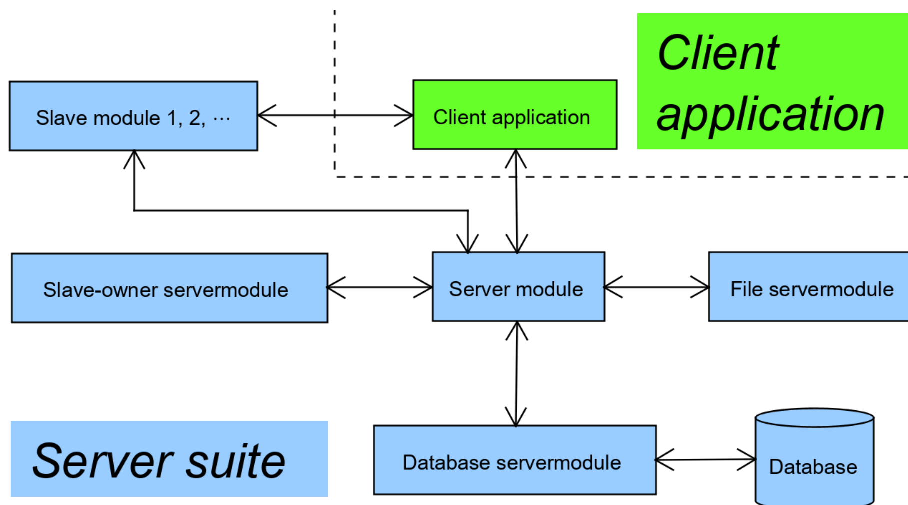


Figure 9: Overall system structure

3.2 Frontend design

This section zooms in on the *client application* from the figure 9. An overall design of the *client application* is shown in figure 10. It is built using Electron and uses a communication module that communicates with the *server suite* using the *generic communication library*. These parts are described individually in the subsequent sections.

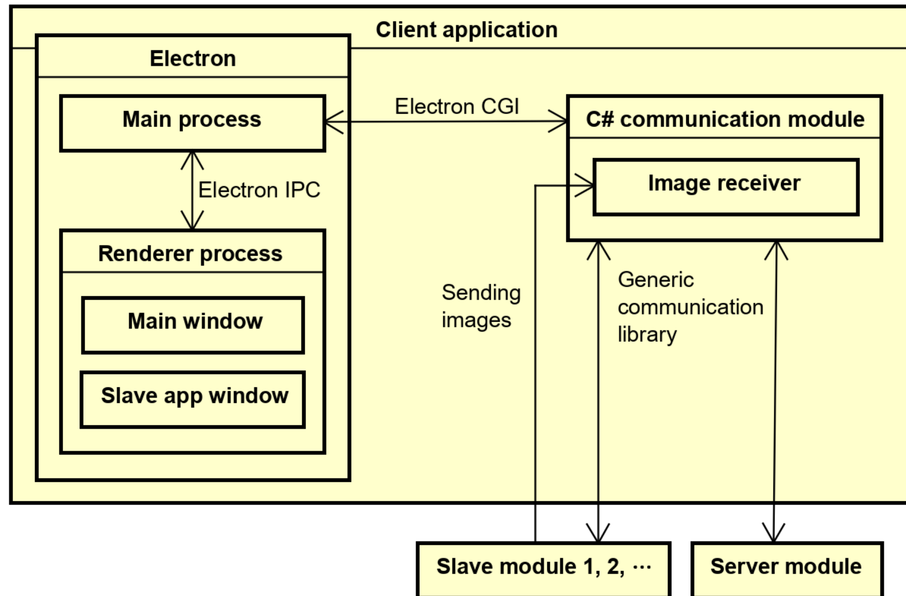


Figure 10: *Client application* overall design

3.2.1 Electron

Electron is a modern framework developed by GitHub (Electron, 2019a) for creating native cross-platform applications with web technologies like JavaScript, HTML, and CSS. It accomplishes this by combining Chromium and Node.js into a single runtime. The resulting app can be packaged for Windows, Mac and Linux. It is used by Slack, Discord, Visual Studio Code and many more applications (Electron, 2019b).

Before diving further into the details, the two process types available in Electron need to be discussed. They are fundamentally different and important to understand.

An Electron app always has only one **main process** which can display GUI by creating web pages. Each web page in Electron runs in its own **renderer process**. The two processes can communicate with each other using Electron Inter-Process Communication (IPC) described in Electron (2019c) and Electron (2019d).

Electron and its main and renderer process and their IPC can be seen on the left side of figure 10. Renderer process is responsible for rendering both the *main window* (which is also the startup window) and the *slave application window(s)*.

The part of the *client application* using Electron is responsible for showing and managing the GUI, but not for communication with the backend.

3.2.2 Communication module

For communication with the backend a communication module is necessary. Due to the usage of *generic communication library* it has to be written in C#. It is responsible for communication with the *servermodules* and *slave module*. This communication is done asynchronously in a callback fashion as described in section 3.3.2 with the exception of receiving images from the *slave module* for which a socket connection is used for the highest efficiency.

The main process in Electron communicates with this communication module via Electron CGI (Common Gateway Interface[6] (Figueiredo, 2019) and, if necessary, relays the information to the renderer via the IPC.

Electron IPC and Electron CGI are using JSON to transmit the data which is simple to work with in both the Electron part and communication module part of the *client application*. Sending images is done by just transmitting the bytes and it is the responsibility of the communication module to receive a full image which is saved in a location from which Electron is continually loading a new refreshed image.

For maintainability purposes, communication module uses NLog (NLog, 2019) heavily to log info from the running program and to ease an identification of a potential problem.

3.2.3 Event capture

The *slave application window* is responsible for capturing mouse and keyboard events as described in section 2.2.3. These are delegated to the main process and sent further to the communication module. From there, they are sent to the *slave controller* where handled by their respective APIs as described in section 3.4.7.

3.2.4 Web technologies

Although an Electron application can be created by using pure HTML, CSS and JavaScript, other web frameworks and technologies can be added. There are many tutorials and boilerplates available that combine usage of Electron with other web technologies (Sorhus, 2019). This tutorial (Vītoliņš, 2019) sets up the project with TypeScript, webpack, React and Electron CGI. Bootstrap and Sass support has been added as well. These technologies are described shortly in the subsequent paragraphs.

React

React is a JavaScript library for building user interfaces and has affected the overall design of the frontend the most with its **components** approach(Facebook Inc., 2019a).

---

[6]This module uses this name as it borrows the main idea from CGI, although it is not a full-blown CGI as described in Robinson and Coar (2004)

Building an application with React means creating a lot of components and creating a logical hierarchical structure from them. These components do not separate concerns such as view and controller as it is usually done in a design pattern such as MVC, but rather combine them in small manageable chunks. Refer to Hunt (2019) for a more detailed introduction to this topic.

TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript (Microsoft, 2019).

webpack

Webpack is used for bundling the source code and other assets (Wikipedia contributors, 2019e). It needs to be run after a change to the source code, as it generates its own output files that are used in the application while running. Webpack is highly configurable and it allows splitting the source code into small logical sections while in the end producing an efficient and standardized code for the application.

Bootstrap

Bootstrap is the most popular CSS framework for web development (GitHub, 2019). The layout utilities and a few components from Bootstrap are used in the project.

Sass

Sass is a preprocessor scripting language that is compiled into CSS (Wikipedia contributors, 2019b). It supports two syntaxes. The project is using the SCSS (Sassy CSS) form which is closer to that of a regular CSS. The project mainly uses variables and nesting features provided by Sass.

3.3 Middleware design

A custom generic remote procedure call middleware is designed for this system. It is designed to use the message queue framework NetMQ, which is a C# adaptation of the more widely used ZeroMQ (ZeroMQ, n.d.).

In section 3.3.1 the NetMQ framework is expanded upon. It is necessary to understand some principles of the NetMQ framework to properly comprehend the design decisions for the *generic communication library*.

In section 3.3.2 the design and design decisions of the *generic communication library* are described. This includes how the middleware solves some of the requirements for the system design.

In section 3.4.1 it is described how the *generic communication library* is used in the design for the communication between the *servermodules* and the *client application*.

In section 3.4.7 it is described how the *generic communication library* is used and extended upon, to design the communication between the *client application* and the *slave module*. However, this is mostly the same as in section 3.4.1.

3.3.1 NetMQ / ØMQ

As has previously been mentioned, NetMQ is a C# adaptation of the asynchronous message oriented message queue library ZeroMQ (ZeroMQ, n.d.). ZeroMQ is a message queue as well as a framework for easier creation of applications that communicate through a network. ZeroMQ is a very extensive library and has many advanced design patters. However, due to the time constraint on the project, the main functionality that is used is its Request and Reply socket.[7] In NetMQ, a socket is an object that behind the scenes can have many network connections. That means a server socket in NetMQ, that is bound to a port on a machine, can be connected to by a very large amount of clients. In fact, the only limitation is the amount of available ports and the computation power to handle a large amount of requests in a timely manner.

When a server socket receives requests from many different clients, it uses a technique called fair queuing, which means a request is handled for each client before a second request is handled for any of the connected clients.

As has already been stated, mainly `RequestSocket` and `ResponseSocket` are used from the NetMQ library. These two sockets represent an active and a passive part of the network connection. The `RequestSocket` is always the one making requests, and a `ResponseSocket` always responds to requests. The NetMQ sockets are made as state machines, which means that a `RequestSocket` cannot send out two messages without receiving a reply to the first request before sending out the second request. The same goes for the `ResponseSocket`, except it must first receive a request and then send back a reply.

The messages that can be sent through these sockets are `NetMQMessages`. These represent a series of `Frames` that together make up the message. NetMQ handles the full transport of a message and the programmer does therefore not need to worry about receiving `n` number of frames and so forth. The data that is sent between two sockets is in byte form. However, NetMQ has wrappers for handling strings without the programmer having to worry about string encoding. How the messages are structured using several frames in the *generic communication library* is further explained in section 3.3.2.

3.3.2 Generic communication library

The *generic communication library* is the basis for all the network communication that happens between nodes. The *generic communication library* can be described as a piece of asynchronous message oriented middleware.

These technical requirements exist for the *generic communication library*:

1. A call to a remote method must happen asynchronously

[7] The sockets in this library do not represent a standard network sockets. The word socket is merely a name that is used, as the design of the framework is inspired by how programmers are used to program network applications (Hintjens, 2019).

2. The library must support routing capabilities. That is, it should be possible for many nodes to communicate using a central node as a message router.

Technical requirement 1 is required so that a single call does not make a module unresponsive. Furthermore, due to a module being unresponsive, CPU time is wasted.

Technical requirement 2 is required so that a deployment of the system can be simplified and for an increased scalability and reduced maintainability.

Sendable objects

Before diving into the design of the *generic communication library*, the objects that can be sent using the *generic communication library* must be described first.

Figure 11 shows the objects that can be sent through the *generic communication library*. There are three classes — `Sendable`, `Response` and `BaseRequest` — each having a singular purpose. The `Sendable` base class's main purpose is to allow polymorphism, however, it also serves to improve maintainability by having less code duplication.

As can be seen in the figure 11, the `Sendable` class contains two fields: `senderModuleID` and `callID`.

The `senderModuleID` is the `ModuleID` of the module initiating a request. A module gets its `ModuleID` by registering itself to a `BaseRouterModule` as can be seen in figure 12, using the method `RegisterModule`. By registering itself to a `BaseRouterModule`, the router module then knows how to send requests and responses to a given module without having to set up connection configuration for each module.

The `callID` is generated by the module that initiates a request. It is used so that the module can know which request a given response is for.

As was mentioned in section 3.3.1, the sockets used for the communication are state machines and cannot send out or receive two messages in a row. It is possible that NetMQ supports our use case directly with a different design pattern. However, due to time constraints these options were not researched. Therefore, a workaround is employed such that every time a message is handed over to another module, an acknowledgement message is sent back, thereby bringing the sockets back in a state where they can send or receive again.

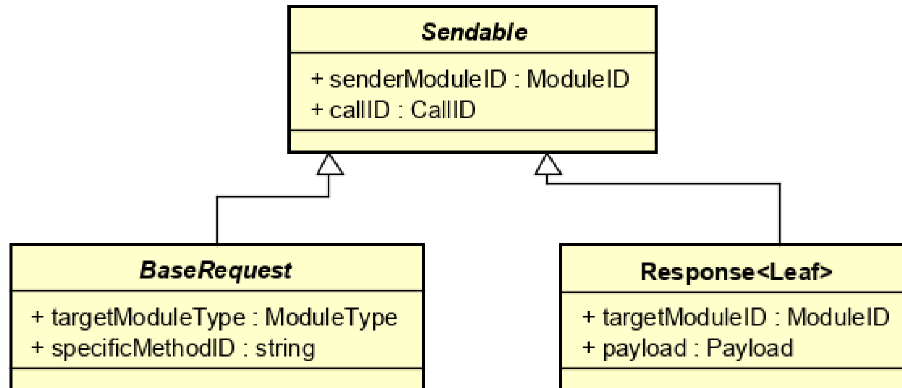


Figure 11: Objects sendable via *generic communication library*

The class `Response` is a leaf class and it allows for delivery of the return object of a remote method call. The return object is stored in a `Payload` class, which is simply a wrapper for an `Object` class. There are two reasons for the `Response` to be a leaf class. First, it makes it easier to deserialize and second, it is not necessary to have type safety because when making a specific remote method call, the return type is already known and therefore typecasting is trivially done.

As the response must always be sent back to the specific module that initiated the request, the `Response` class has a `targetModuleID` field, which is used by the `BaseRouterModule` to forward the response to its destination.

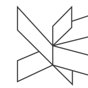
The `BaseRequest` abstract class is made to be inherited from, when creating a communication library that is based on the *generic communication library*. Any class that inherits from `BaseRequest` is treated as a request that can be sent to other modules.

Because there is a number of unknown classes inheriting from `BaseRequest`, it is necessary to specify to the library how to deserialize a `BaseRequest` into the required subclass. This is done by inheriting from the `Encoding` class seen in figure 12, which is why the method of the `Encoding` class is abstract.

The `BaseRequest` class has a `targetModuleType` field, whereas the `Response` class has a `targetModuleID`. This is because a request can be handled by any node as long as it is of the right type, whilst a response needs to be sent back to the specific module that initiated the request. Furthermore, this way, a module making a request does not need to know the specific `moduleID` of the module that it makes a request to.

Generic communication library architecture

Next, the overall structure of the *generic communication library* is described. Figure 12 aids in the description of the *generic communication library*.



Encoding

DecodeJsonToSpecificRequest(specificMethodID : string, jsonString : string) : BaseRequest
+ EncodeSendable(sendable : Sendable) : NetMQMessage
+ DecodeIntoSendable(message : NetMQMessage) : Sendable

1

ProxyHelper

+ moduleID : ModuleID

+ SendResponse(response) : void
+ Setup(...) : void

1 * 1 1 1

BaseCommunicationModule

CALL_ID_PREFIX : string
+ ModuleType : ModuleType

+ Setup(...) : void

0..* 1

BaseProxy

- callBacks : Dictionary<CallID,Action<Response>>

+ HandleResponse(response : Response) : void

BaseServerModule

+ handleRequest(request : BaseRequest) : void

BaseClientModule

BaseRouterModule

MODULE_ID_PREFIX : string
+ moduleIdToRequestSocket : Dictionary<ModuleID,RequestSocket>
+ moduleTypeToRequestSocket : Dictionary<ModuleType,RequestSocket>

+ ForwardSendable(sendable : Sendable) : void
+ RegisterModule(moduleType : ModuleType, ConnectionInformation : int) : void

Figure 12: Reduced class diagram for the *generic communication library*

There are three types in this library that any communicating entity can be — `BaseRouterModule`, `BaseServerModule` and `BaseClientModule` — each being a specification of the `BaseCommunicationModule`.

The `BaseCommunicationModule` has few responsibilities, but it is still important. For brevity's sake, its constructor has been omitted in figure 12. However, it takes a `ModuleType` as an argument.

The `BaseClientModule` is a type of module that can only make requests and get responses. Therefore any module that extends this class represents a client in a client-server like communication.

For a module to handle incoming requests, a module must inherit from `BaseServerModule`, which can both make requests of its own, and also respond to requests made to it.

The `BaseRouterModule` allows other modules to register to it and forwards both requests and responses it receives that are not intended for itself.

In the case of responses, the router module uses the `targetModuleID` and looks up a connection information for that specific module and forwards the `Sendable` object there.

In the case of a `BaseRequest`, the `BaseRouterModule` looks up all registered modules of the same type as the `targetModuleType` and then at random[8] forwards the request to one of these modules.

In order to have the responsibility of sending to and receiving from the network at a singular location, the `ProxyHelper` was added. Each module has at least one instance of `ProxyHelper`. When the setup method of a `ProxyHelper` object is called, it registers itself to the `BaseRouterModule` that it has been given the connection information for. This results in the `ProxyHelper` receiving a `ModuleID` from the `BaseRouterModule`. As has already been stated, this `ModuleID` is then used as the sender `moduleID` for any requests made with this `ProxyHelper`.

As has already been explained in section 3.3.1, NetMQ works by sending `NetMQMessages`. However, to be able to meaningfully serialize and deserialize a message, some standard for how a given `Sendable` object is encoded to and decoded from a `NetMQMessage` must be made.

The standard used in this library for encoding and decoding `Sendable` object is the following. Every `NetMQMessage` must have exactly two `NetMQFrames`. The first frame must contain a string that is either the string literal "RESPONSE", or a string that can be used to unambiguously identify the request type, which means a subclass of `BaseRequest`. The second frame must include an object encoded to a string using JSON. The string from the first frame can then be used to figure out how to deserialize the JSON object that is stored in frame two. Having discussed how the encoding works, it is now relevant to discuss how a request can be made.

A request can be made using a class that inherits from the `BaseProxy`. The `BaseProxy` serves as a basis for implementing proxies that can be used by any given module to call remote methods on a specific module. The `BaseProxy` holds some protected methods that simplify the code that has to be implemented in a class that inherits from the `BaseProxy`. The `BaseProxy` requires a `ProxyHelper` when it is created because it needs the `ProxyHelper` to send out the `NetMQMessages`.

[8] In production the load-balancing would be more advanced

Furthermore, the purpose of having proxies is to hide some of the complexity in the creation of the `Sendable` objects. Therefore, for every module inheriting from the `BaseServerModule`, a proxy should be implemented. The proxy should naturally inherit from the `BaseProxy`. This design is made so that the functionality of these modules can be consumed easily by any other module that inherits from the `BaseCommunicationModule`. Finally, when a response comes back for a request, it is the proxy that is responsible for activating the correct callback method.

Callbacks

When calling a method on a proxy, both the arguments as well as a callback method must be provided.[9] Callbacks are used to handle an asynchronous communication in the *generic communication library*. One might ask why the more modern C# async and await code pattern was not used and there are two reasons for this. First, it is only available in a pre-release version of the NetMQ library. Second, it is due to time constraints combined with the fact that the responsible developer has no previous experience with await and async code pattern.

3.3.3 Usage in distributed system

Figure 13 aids in understanding how the *generic communication library* is used in a distributed system. For the sake of simplicity, a lot of information is left out. The figure should be read in a way that each package is its own process, each potentially running on a different machine. It can then be seen how the `BaseClientModule` uses the `BaseProxy` to call a remote method on the `SlaveOwner`. The `BaseClientModule`'s `BaseProxy` uses the `ProxyHelper` to send a remote method call object to the `ServerModule` using the `RequestSocket`.

When the remote method call object is received on the `ServerModule`, it is recognized as a request that needs to be forwarded, which is done by picking the `RequestSocket` that points to the `SlaveOwner` and sending the object there.

In the `SlaveOwner` it is identified as a request that needs to be handled and is given to the `BaseServerModule` by the `ProxyHelper`. When the request is processed, the response object is sent to the `ServerModule` using the `RequestSocket` of the `SlaveOwner`. Then again, the `ServerModule` identifies the object as a response, picks the `RequestSocket` that can send to the module the response is designated for, and sends the response there.

When the response arrives back at the `ProxyHelper` of the module that made the request, the `ProxyHelper` first identifies which `BaseProxy` is responsible for the request with the given `CallID`. Then the response is given to the corresponding `BaseProxy` and here the callback that is connected to the `CallID` is executed using the response object.

[9] Instead of providing a callback, another option is to implement the method so that it polls on a private variable in the proxy until a response arrives and then returns this value. This would make the method synchronous. However, it is important to mention that timing out should be implemented such as not to wait for a response forever.

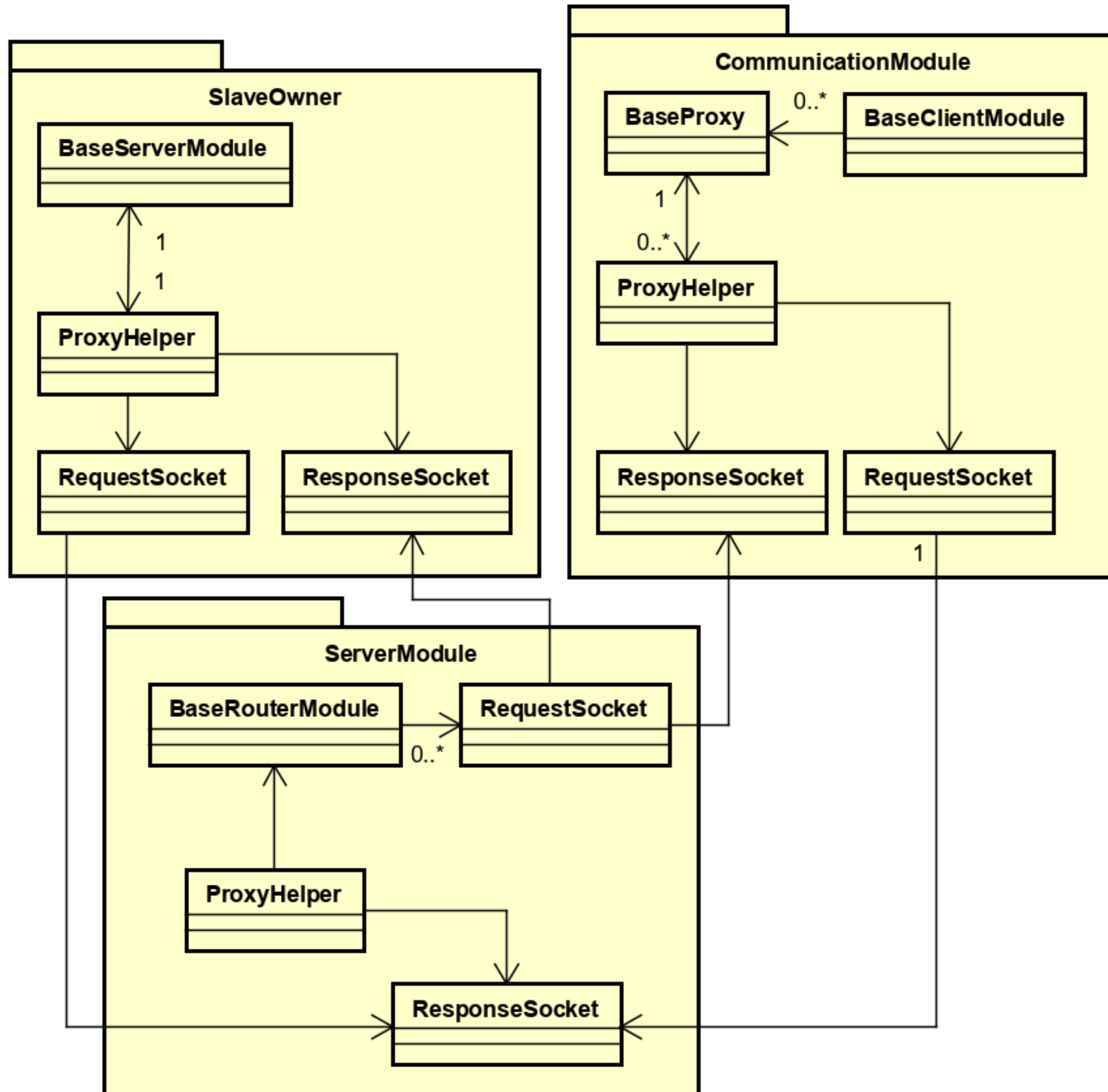


Figure 13: Simplified overview of the connections in the distributed system

3.4 Backend design

The design of the backend revolves around the entities that are responsible for the business logic, referenced to as *servermodules*. Each *servermodule* has a single area of concern. This separation has been done with consideration to scalability. One *servermodule* is responsible for handling the virtual machines that run the applications (section 3.4.4), another is responsible for handling files (section

3.4.5) and the last one is responsible for database access (section 3.4.6). Furthermore, there is also a `SlaveController`, which represents the business logic running on *slave modules*. However, first an overview of the backend design is given in section 3.4.1, followed by a description of Docker in section 3.4.2. Docker is described as all of the *servermodules* run inside Docker containers.

3.4.1 Backend design overview

It can be seen in figure 14 how the modules of the backend are using the *generic communication library*. Besides the classes shown in the figure, each class below the line, which inherits from the `BaseServermodule` either directly or indirectly, needs a proxy. In addition, many request classes that inherit from the `BaseRequest` have been implemented, as well as some model classes.

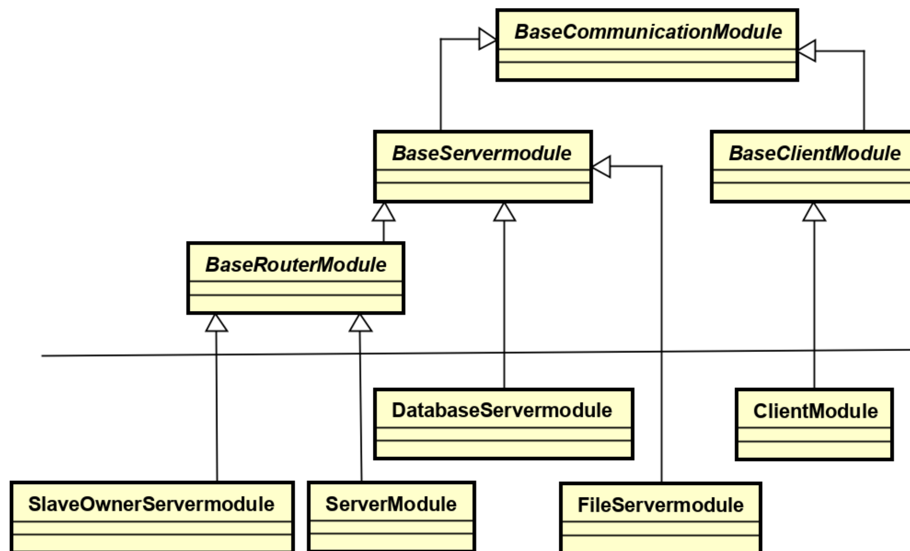


Figure 14: Client-to-servermodules communication library inheritance overview

3.4.2 Docker

Docker is a containerization framework (Docker Inc., n.d.[b]). This means that each application[10] in Docker runs in its own container which simulates a constant environment with all the required dependencies and consistent system settings. This ensures that the application always runs the same no matter where the Docker container is hosted. In this way, containerization simplifies deployment and management of applications.

Docker is used to run all of the *servermodules*. It was originally intended to also use Docker for the *slaves*, it was nevertheless discovered that Docker is not

[10] Must be a console application and not a GUI application

intended for GUI application containerization. The *slaves* are therefore realized as virtual machines. A technique that could be used instead of using virtual machines is App-V (Microsoft, 2018) or X11 forwarding (Business News Daily Staff, 2018). However, these were not further explored.

A major reason for using Docker is that it provides a consistent environment for an application. This eliminates a possible failure point for an application and is important especially in production.

The Docker container image also provides a simple package containing everything that the application needs which simplifies work on different part of the project as it abstracts away the complexities of the already developed parts of the application in a Docker container (Red Hat, n.d.).

3.4.3 Server module

As has been stated earlier, the `ServerModule` has the purpose of routing messages to ease deployment, reduce maintenance and increase scalability. However, as these are not major concerns in this project, there are no noteworthy design decisions to be described.

3.4.4 Slave-owner servermodule

The responsibility of the *slave-owner servermodule* is, as can be deduced from its name, to manage the active slaves and to boot up more slaves if necessary. However, as this system is only a simplified version of the production ready system, the `SlaveOwnerServermodule` does not dynamically boot more slaves.

The `SlaveOwnerServermodule` is initialized with the information (including network information) for the slaves, using system arguments, and the slaves are started manually.

When the `SlaveOwnerServermodule` is initialized, it is responsible for handling requests made to it. First request that can be made to a `SlaveOwnerServermodule` is `GetSlave`, which returns an object that contains the necessary network information for the *client application* to connect to a given *slave module*. When requesting a `Slave`, the primary key of the calling client as well as which application the *slave module* should be running must be supplied.

The second request is a `GetListOfApplications` request that returns a list of all the applications that are supported. This information is used to show a list of applications on the *client application* as shown in figure 5.

3.4.5 File servermodule

The area of concern for *file servermodule*, is to manage everything that has to do with file storage and file access in the system. First, the file ownership is discussed followed by a description of the supported remotely callable methods.

File ownership

As the system is designed now, the ownership of the files on the *file servermodule*

is stored by saving files from a given user into a folder with that user's primary key as the folder name.[11] In a production ready system, the ownership should be stored in a database. This approach allows for more flexibility and control and makes it easier to add features such as sharing files.

Remote callable methods

Firstly, the *file servermodule* has a method called `UploadFile`. When the *file servermodule* receives a request of this type, the request already contains the file data as a byte array and the filename as a string. The directory location for the user is based on the primary key that is also sent as an argument. Then *file servermodule* checks if a file of that name already exists and if it does it checks the last parameter of the remote call to check if it should overwrite the file or not.

The second method is `GetListOfFiles`. It is called from the *client application* and it simply returns a list of all the files that the logged in user (the only parameter) owns which are displayed to the user as shown in figure 6.

The method `DownloadFile` simply downloads a file from the *file servermodule*. The parameters for this remote call is the file name and the primary key of the user.

The method `RemoveFile` takes same parameter as `DownloadFile` and removes the given file from *file servermodule*.

Lastly, the method `RenameFile` renames a given file to a new file name of the logged in user which represent the three parameters of this remote method call.

3.4.6 Database servermodule

The `DatabaseServermodule` is an instance of `BaseServermodule`, which means it serves requests made by the `DatabaseServermoduleProxy`, an instance of `BaseProxy`. The specific requests handled by `DatabaseServermodule` are `LoginRequest` and `CreateAccountRequest`. These requests query or insert into a database described in the subsequent section.

Database

The database stores only login information of a user in a form of email and cleartext password (Cornell, 2007). The database therefore has only a `User` table as shown in figure 15.

Password is stored in cleartext as all security precautions are delimited. Database stores only users at this point but it is easy to imagine it could store information about the files, applications and *slaves* running the applications.

[11] This is obviously not optimal from a security perspective. However, all security concerns in this system are delimited.

User

UserID	INT	NOT NULL
Email	VARCHAR (256)	NOT NULL
Password	VARCHAR (256)	NOT NULL

Figure 15: ER diagram showing the `User` table

3.4.7 Slave controller

The *slave controller* runs on a *slave* and it is responsible for controlling it. The `SlaveController` controls the *slave* by accepting commands from the *client application* and forwarding these commands to Python processes that can execute them using the PyAutoGUI API.

The *slaves* are setup as virtual machines that must be booted manually. These virtual machines are run using Hyper-V, which is a default virtualization software that comes with Windows (Pro/Education).

Communication between client application and slave module

The communication between the *client application* and the *slave module* uses the *generic communication library* and its design can be seen in figure 16. It has been decided that the `SlaveController` inherits from `BaseRouterModule`. This has been done because any communication using the *generic communication library* requests must have a module ID and to get a module ID in the way the library is designed, it must be a `BaseRouterModule`. Since the `ClientModule` is the active part and does not receive requests, the `SlaveModule` was chosen to be the `BaseRouterModule`.

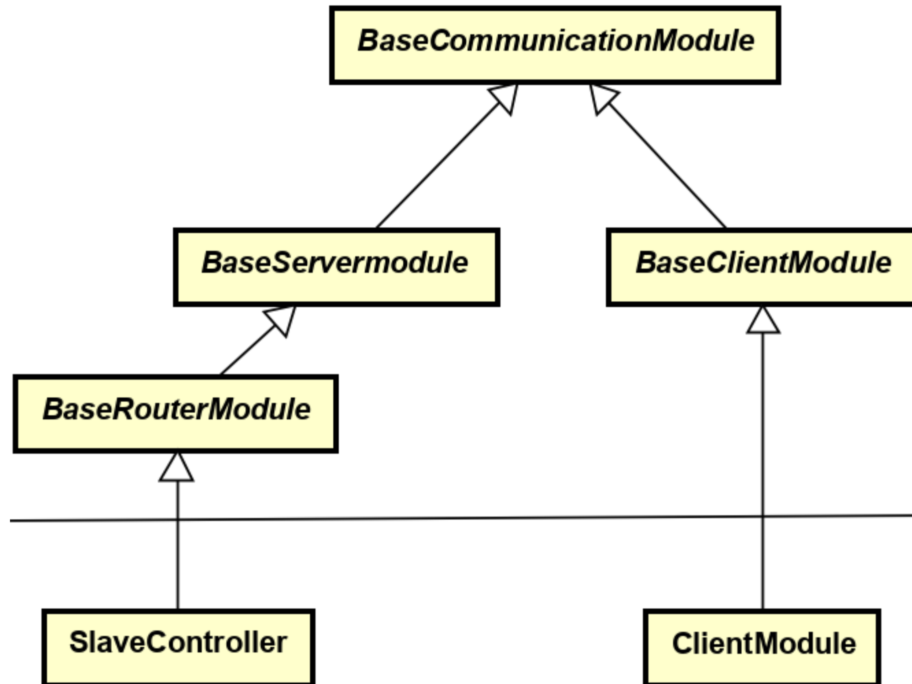


Figure 16: Client-to-slave communication library inheritance overview

Remotely callable methods

The `SlaveController` has a few remotely callable methods that are discussed in this section.

The first method `Handshake` takes a primary key as a parameter. The primary key is then stored by the `SlaveController` to use in future calls to the system on behalf of the user connected to the `SlaveController`. The method then returns a window size of the application running on a *slave module* that is used to adjust the size of the *slave application window*.

The method `DoMouseAction` triggers a specific mouse action on the *slave module* based on the given parameters.

The method `FetchRemoteFile` is called to get a *slave module* to download a file from the *file servermodule* as to make a file from the *file servermodule* available to the application running on the *slave module*.

Lastly, the method `SaveFilesAndTerminate` is intended to be called when the *client application* is done using the *slave module*. The `SlaveController` upon receiving this method call saves the files that have been downloaded or updated to the *file servermodule* and then terminates.

PyAutoGUI

PyAutoGUI is described first as it is used in many of the APIs described in the following sections. Since .NET Core was chosen (see section 2.6), it is necessary to

use another technology for working with the screen, mouse and keyboard. This is the case because .NET Core does not assume the existence of aforementioned input/output devices. The technology that is chosen to handle these capabilities is Python using the PyAutoGUI library (Sweigart, 2019).

PyAutoGUI is a library that is intended for automating GUI tests, and therefore has capabilities for screen capture, mouse and keyboard control. The communication between the Python process and the `SlaveController` is realized with a standard TCP connection, sending commands and arguments as strings. For simplifying the startup process, the `SlaveController` is responsible for starting the Python processes.

Mouse control

The mouse control API supports the *need to have scenarios* actions specified in section 2.2.3. It does this by adding a single remotely callable method to the *slave controller*. This method accepts an object as an argument. By using this argument it is possible for the *slave controller* to determine which of the mouse actions listed below should be executed:

- Left mouse button down
- Left mouse button up
- Right mouse button down
- Right mouse button up
- Mouse move

This method then queues up the mouse action that is to be sent to the Python process that runs PyAutoGUI for controlling the mouse, when it is ready for the next action.

Keyboard control

Controlling the keyboard on the *slave module* is done using PyAutoGUI, which supports both key down and key up events. For a *slave application window* to send a key down or key up command to the *slave module*, there is only a single remotely callable method called `DoKeyboardAction`. This method takes a string that represents the key and a boolean indicating if the action is key down or key up.

When the *slave controller* receives a `DoKeyboardAction`, it stores it to a queue of keyboard actions to execute and a response is sent back to the *slave application window*. Running in a separate thread is a loop that continuously dequeues items from the queue of keyboard events and sends the dequeued item, using a TCP connection, to a Python process that executes the command.

Screen capture and image sender

Screen capture can be approached from multiple stages of implementation difficulty. A standard solution without any proprietary technology is streaming

a video encoded by one of the most used codecs (such as VP9, H265, AV1 (Wikipedia contributors, 2019d)) with WebRTC (Mozilla Contributors, 2019b). Streaming a compressed video as compared to streaming compressed images is always going to be more efficient in terms of the data necessary for a certain quality because of the additional processes that can be applied only to a video, such as block motion estimation (Isikdogan, 2018). Using WebRTC seems to be the best choice to achieve the lowest latency (Unreal Streaming Technologies, 2019). The act of capturing the desktop screen and encoding it to a video can be done by a tool such as FFmpeg (FFmpeg contributors, 2019). Additionally, this solution could also easily bundle the audio stream from the application on the *slave module* as a part of the video.

However, this solution requires a specific knowledge and quite a lot of resources. Therefore a more straightforward solution of using PyAutoGUI to take screenshots as JPEGs and sending these to the *client application* was implemented.

In the production ready system, technologies such as App-V or X11 forwarding could be used for interacting with the GUI running on the *slave module*. However, due to the scope constraints on this project, there simply has not been enough resources to look into the viability of these technologies.

To reduce latency and increase throughput of images, the Python image capture sends the images directly to the *client application* through a regular TCP connection. The *client application* gets the connection information to the Python image capture process from the *slave-owner servermodule* and connects to it. The protocol for sending the images through the TCP connection is then as follows. First, an integer, containing the size of the image that is to be sent, is converted into bytes and those are then sent through the TCP connection. Next, the image is read into memory as a byte array and is sent via the TCP connection as well. For receiving the images, the reverse process should naturally be followed. That is, first read the bytes corresponding to an integer and use the integer value to determine how many bytes to read. Now read the bytes and then save them to a file. More about the saving process follows in section 4.1.1.

4 Implementation

This section shows a few code listings from the project related to a certain topic and describes them in detail. For brevity's sake, the code listings are shortened. Full source code can be found in appendix B.

4.1 Frontend

The frontend implementation describes image receiver in detail and shows a typical usage of React in the project.

4.1.1 Image receiver

This section focuses on the image receiver from figure 10, specifically the usage of a regular TCP socket for performance considerations and how are the images received and used in the *slave application window*.

The communication module uses a TCP connection for receiving images. This is the only place within the project where a low level TCP connection is used directly. The reason to use a TCP connection here is for better performance. However, this entails working with the byte buffers and designing the protocol in such a way as to ensure a valid image transmission. The image size needs to be transmitted first so that a byte buffer of an appropriate size can be instantiated and afterwards filled with the image data.

The image data then needs to be stored to an image file which the *slave application window* can show. The *slave application window* has an interval set which refreshes the image as shown in code listing 1. This means that the *slave application window* can read the image at any time, even when the image is being written to by the image receiver which can result in the *slave application window* showing a broken image icon shown in figure 17.

```
setInterval(() => this.updateImage(), 50); // time in ms
```

Listing 1: *Slave application window* interval for updating an image

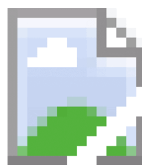


Figure 17: Broken image icon (McKalin, 2018)

To prevent this, the image data are stored from the byte buffer to a buffer file and only when this is complete, this buffer file is copied to an image file that is being shown by the *slave application window*. Note that this does not fully solve

the problem as the *slave application window* can still read the image file when the buffer file is being copied over. However, it reduces the chance of the *slave application window* reading the image file while it is in a non-readable state and would produce the broken image icon.

4.1.2 React

React greatly facilitates building of single-page applications. As Facebook Inc. (2019c) describes, by updating the state variable, React re-renders the component. Furthermore, as described in Facebook Inc. (2019b) it only updates part of the DOM that has been changed. As an example, the *main window* decides whether to render a login form (figure 3) or an applications view (figure 5) based on a `loggedIn` state variable. When it is changed, React re-renders the view according to the code listing 2.

```
1 var toRender;
2 if(this.state.loggedIn) {
3   toRender = this.GetAfterLoginView();
4 } else {
5   toRender = this.GetLoginView();
6 }
```

Listing 2: Selective rendering based on `loggedIn` variable

Feature flag

As it is written in the code listing 2, login form is always going to be shown first when the application launches. In the case when a developer is not working on a login functionality, he does not need to see it. For that case a feature flag (Wikipedia contributors, 2019a) that can disable the login functionality has been implemented and used as is shown in the code listing 3.

```
1 var toRender;
2 if(FeatureFlags.AllowLogin) {
3   if(this.state.loggedIn) {
4     toRender = this.GetAfterLoginView();
5   } else {
6     toRender = this.GetLoginView();
7   }
8 } else {
9   toRender = this.GetAfterLoginView();
10 }
```

Listing 3: Usage of a feature flag

4.2 Middleware

This section covers some interesting implementation that is based on section 3.3.

4.2.1 Encoding

To be able to send responses as well as requests between the different nodes in the system, these messages need to be serialized and deserialized. As has already been discussed, the communication between the nodes is handles by NetMQ. However, NetMQ can only handle objects of type `NetMQMessage`. The purpose of the class discussed in this section is therefore to serialize and deserialize object to and from `NetMQMessage` object, respectively.

In code listing 4, the `Encoding` class can be seen with an abstract `DecodeJsonToSpecificRequest` method and two concrete — `DecodeIntoSendable` and `EncodeRequest` — methods. The abstract method is called from the. `DecodeIntoSendable` method.

```
public abstract class Encoding
{ // several methods are hidden

protected abstract BaseRequest DecodeJsonToSpecificRequest(
        string specificMethodID, string jsonString
    );

public Sendable DecodeIntoSendable(NetMQMessage message)
{
    var first = message.Pop().ConvertToString();

    if (RESPONSE_PREFIX.Equals(first))
    {
        return TryDecodeJson<Response>(
            message.Pop().ConvertToString()
        ) ;
    }else if (ACK_RECEIVE.Equals(first))
    {
        return TryDecodeIntoAckReceived(message);
    }
    else
    {
        return DecodeJsonToSpecificRequest(
            first, message.Pop().ConvertToString()
        );
    }

    public static NetMQMessage EncodeRequest(
            BaseRequest request
        )
```

```
31      {
32          var message = new NetMQMessage();
33          message.Append(
34                  new NetMQFrame(request.SpecificMethodID)
35              );
36          message.Append(
37                  new NetMQFrame(EncodeToJson(request))
38              );
39          return message;
40      }
41  }
```

Listing 4: Sample of the `Encoding` class

`EncodeRequest` is used to build an object of the type `NetMQMessage` from an object of a class that inherits from `BaseRequest`. As can be seen in the code listing 4, the building of the message is done by creating the message object and then appending two frames. The first frame is built using a string that is unique to the type of the request. The second frame is the request object encoded into JSON. Having finished building the `NetMQMessage`, it is returned and sent to another node of the system.

`DecodeIntoSendable` is used on the receiving end, so its purpose is to decode the `NetMQMessage` into an object of a class that inherits from `BaseRequest`. The only assumption made by this method is that the `NetMQMessage`, that is passed as an argument, contains data for an object of type `Sendable`. From here it checks the content of the first frame. The content is used to discover whether this message contains data for an object of type `Response` or `BaseRequest`. If an object of type `Response` is contained, it is decoded. Otherwise, the method `DecodeJsonToSpecificRequest` is called. This method is implemented in a class inheriting from `Encoding` and here a switch determines the specific type of the `BaseRequest` using the string of the first frame, and creates an object of the found type from the JSON in the second frame.

4.2.2 Middleware library

This section discusses some of the implementation that is used in the invoking of remote requests and handling the response that comes back.

First, code listing 5 shows the code that is required to make a remote method call available in a proxy. The necessary code to make a method callable remotely is quite simple as shown.

The request is represented by a class (`RequestGetListOfFiles`) for which arguments needed for the remote method call must be set. In this case the `PrimaryKey` field needs to be specified.

Furthermore, in code listing 5, three methods are called: `SetStandardParameters`, `WrapCallBack` and `SendMessage`. `SetStandardParameters` is a method that just sets some arguments used in routing and is therefore not that useful to have a deeper look at. However, the other two methods are described in

more depth. Furthermore, the method ReceiveSendable is also elaborated on. However, this method is not called in code listing 5.

```
public void GetListOfFiles(PrimaryKey pk, Action<List<
    FileName>> callBack)
{
    var request = new RequestGetListOfFiles();

    request.PrimaryKey = pk;

    SetStandardParameters(request);

    var wrappedCallback = WrapCallBack<List<FileName>>(
    callBack);

    base.SendMessage(wrappedCallback, request);
}
```

Listing 5: Implementation of proxy method for remote method invocation

The method WrapCallBack that is shown in code listing 6, is the first method to be discussed. To clearly understand the purpose of this method, it helps to look back at code listing 5. Here it can be seen that when calling a remote method through a proxy, a callback method must be provided. This callback is an Action and the generic type parameter is of the type that is returned by the remote method. To store these actions easily, they need to have the same generic type parameter. This is where the method WrapCallBack comes into play. This method takes an Action object with any generic type parameter, as long as this inherits from a class and returns an Action object that takes a Response object as argument. WrapCallBack method typecasts the Payload from the Response into the type needed by the original Action and invokes that original action with the typecasted argument.

```
//method found in class BaseProxy
protected static Action<Response> WrapCallBack<T> (
        Action<T> callBack
    ) where T : class
{
    return
        (response) =>
        {
            var thePayload = response.Payload.ThePayload;
            if (thePayload is JArray _jArray)
            {
                response.Payload.ThePayload =
                        _jArray.ToObject<T>();

            }
```

```
16                 else if(thePayload is JObject jObject)
17                 {
18                     response.Payload.ThePayload =
19                             jObject.ToObject<T>();
20                 }
21 
22                 T obj = response.Payload.ThePayload as T;
23 
24                 if(null == obj)
25                 {
26                     throw new Exception(); // shortened
27                 }
28                 callBack.Invoke(obj);
29             };
30         }
31 }
```

Listing 6: `WrapCallBack` method

The code listing 7 shows the method `SendMessage`. It is the last step before a message is sent off to another system node. The `SendMessage` method adds the `CallID` with an object of type `BaseProxy`. This is done so that when a response arrives at the `ProxyHelper` it can use the `CallID` to lookup which object should receive the response object. After this the method simply encodes the request to a `NetMQMessage`, and sends that object off to the `BaseRouterModule` that this `ProxyHelper` is connected to.

```
1  //method found in class ProxyHelper
2  public void SendMessage(
3         BaseRequest message
4         , BaseProxy baseProxy
5      )
6  {
7      callIDToResponseHandler.Add(
8         message.CallID.ID
9         , baseProxy
10     );
11 
12     var req = Encoding.EncodeRequest(message);
13     this.outTraffic.SendMultipartMessage(req);
14 } // shortened
```

Listing 7: Method `SendMessage`

The method `ReceiveSendable`, which can be seen in code listing 8, is called from a thread, that is only responsible for running this method. Because this method is the only code that this thread runs, it is also the reason why there is an infinite loop. First, the method binds a `NetMQResponseSocket` to a port. From here the infinite loop starts. Next, receive an object which can either represent a request or a response. This code listing only shows code relevant to the response

part. Here it is checked if the module receiving the response is a router module and if the response is meant to be received by another module. If this is the case, the object is sent off to another method to be routed to its destination. If the response has arrived at the right location, the `CallID` is used to lookup which `BaseProxy` holds the callback for this response and then the object is forwarded there. From here the callback is invoked, and this is how the remote method invocation flows in a `BaseCommunicationModule`.

```
1 //method found in class ProxyHelper
2  public void ReceiveSendable(
3          Encoding customEncoding
4          , Port portToListenForIncommingData
5      )
6 { // method is shortened
7      ResponseSocket inTraffic =
8              new ResponseSocket(
9                  "tcp://" + "0.0.0.0:" +
10                 portToListenForIncommingData.ThePort
11             )
12     ;
13
14     while (true)
15     {
16         //receive message
17         var message =
18                 inTraffic.ReceiveMultipartMessage();
19         var sendable =
20                 customEncoding.DecodeIntoSendable(message);
21
22         if (sendable is BaseRequest _request)
23         { // shortened, relevent when receiving requets
24         }
25         else if (sendable is Response response)
26         {
27             if (baseModule is BaseRouterModule _router
28                 && response.TargetModuleID.ID != ModuleID.ID
29                 )
30             {//routing
31                 _router.HandleSendable(response);
32             }
33             else
34             {
35                 var id = response.CallID.ID;
36                 if (callIDToReponseHandler.ContainsKey(id))
37                 { //response to a known request
38                     callIDToReponseHandler[id].
    HandleResponse(response);
39                 }
40             }
```

```
41            }
42        }
43 }
```

Listing 8: Method `ReceiveSendable`

5 Test

The desired level of testing for this project is positive testing (GURU99, n.d.[a]), due to the nature of this project being to develop a prototype. Positive testing means that the desired outcome of the test is to show that it is at least possible to achieve the goal of a given *use case scenario*. Almost no effort was spent on negative testing (Nadig, 2019), which would be trying to break the program and to find out how robust the application is with managing faulty input.

In the ideal case, the testing should follow the testing pyramid (Vocke, 2018), which means most tests are unit tests, fewer tests are broader reaching integration tests and the least amount of tests are automated GUI tests. There are two main reasons why no time has been spent on developing the automated tests.

First, the project in its scope is still fairly small and short. Spending time on automated tests makes sense only in the long term. This is because the main value gained from automated tests is to verify that changes to code do not break existing functionality and meet a certain quality.

Second, the nature of this project is such that most of the sensible testing is in form of integration testing requiring more complex setup. Such an automated integration testing is time demanding to create. Since they are more costly to create and the value gained from them is relatively small, they are not done in this project. By comparison, the individual units of the project, for which unit tests could be written, are simple. As such, the tests in this project only consist of manual (user acceptance) tests (GURU99, n.d.[b]).

The setup while testing is that everything runs on the same physical computer. The *servermodules* are launched together using Docker Compose.[12] The *slave module* is running in a Hyper-V virtual Windows 10 machine and the *client application* is running on localhost.

5.1 Test of use cases

In this section, the tests for each of the *scenarios* from all of the *use cases* from section 2.2 are described. The steps of the *scenarios* are used as a test specification, and the post-condition is used as the expected observable outcome.

The *use cases* have *need to have* and *nice to have* parts. The *need to have* parts are tested in a structured manner, using the *use case scenario* steps. The *nice to have* parts are tested in an exploratory manner.

[12] Docker Compose, is a way to run several Docker images "together" where routing is done with IDs instead of static IPs like in DHCP (Docker Inc, n.d.[a])

Below is a test of a full *use case* that has been copied from appendix A to showcase how the testing of the *use cases* has been done.

A table of all the test results (excluding the results of *nice to have scenarios*) can be seen in a table in section 6.1.

Test of use case: Control a running application

This *use case* has 2 defined *need to have scenarios*, which are tested one by one.

Test of scenario 1 - Use of left and right mouse buttons, both up and down events

Precondition: Having launched a specific application
Expected outcome: See that the mouse events were activated

Test steps:

1. Hover the mouse above the *slave application window*
2. Press down on either left or right mouse button
3. Optional: move the mouse
4. Release the mouse button to activate the up event

Executed steps:

1. The application Paint is used for this test
2. Move the mouse above the Pencil tool
3. Press left mouse button down
4. Release left mouse button
5. Observe that the tool got selected
6. Move the mouse such that it is above the "Home" tab
7. Press right mouse button down
8. Release right mouse button
9. Observe that the context menu appears

Observed outcome: It was observed that both the left and the right mouse button actions occurred.

Test of scenario 2 - Keyboard control - Character keys, enter and backspace, both down and up events

Precondition: Having launched a specific application and being in a state where typing on the keyboard produces an observable outcome

Expected outcome: See that the expected key output occurred

Test steps:

1. Press key
2. Release key

Executed steps:

1. The application Paint is used for this test
2. The Text tool is selected from the toolbar
3. Create a text field by left clicking on the canvas
4. Enter the following pieces of text to demonstrate that every character key is working
    (a) " the quick brown fox jumps over the lazy dog " (Wikipedia contributors, 2019c)
    (b) " THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG "
    (c) " 1234567890-= "
    (d) " !@#$%^&*()_+ "
    (e) " []|<>{}\;’,./:"?`~"
5. Furthermore, <Enter> and <Backspace> were also tested using the Text tool by typing <Enter> to create a new line and <Backspace> to delete the new line

Observed outcome: All of the keys worked and the test can therefore be deemed a success

Test of nice to have - Control a running application:

1. Continuous mouse position update
    - This was tested by drawing a circle in Paint. It was found that a circle can be drawn. However, it appears that the mouse position is not updated that often and therefore the circle tends to look more like a polygon.
2. Scrolling
    - Using WordPad as the test application it was found that scrolling does not work
3. All remaining keyboard keys, both down and up events

- The following keyboard commands were tested in WordPad:
  - <Ctrl>+<S>, triggered a save dialog window
  - <Page Up> worked
  - <Tab> worked
  - <F10> worked
  - <Insert> worked
  - <Delete> worked
  - <Esc> worked

4. Resize the *slave application window*
   - It was not found possible to resize the *slave application window*.
5. Changing the local cursor so it matches the one on *slave module*
   - Did not occur during testing

5.2 Test of non-functional requirements

Besides *use cases* with *scenarios*, there are also non-functional requirements for this system. As with the test of the *use case scenarios*, the test results can be seen in a table in section 6.1. The tests of non-functional requirements are described below.

5.2.1 Non-functional requirement 1 (Windows 10)

The first non-functional requirement to be tested is "the *client application* must run on Windows 10". This is somewhat trivial to test. The step to do this test is to be on a Windows 10 machine and try to run the *client application*. This test completed with a success.

5.2.2 Non-functional requirement 2 (CPU utilization)

The non-functional requirement that is tested here is "the *client application* must be able to run on a low-end laptop CPU with an average CPU Mark score of 4967 (PassMark, 2019) with three concurrent *client applications* running, never exceeding 30% utilization".

This test is conducted on a computer with a CPU having exactly the CPU Mark score specified in the non-functional requirement. This computer runs only the *client application*. Another computer runs all of the *server suite*, including the three *slave modules*. The applications that are used here are two instances of Paint and one instance of WordPad.

The result of the test was that running of the *client application* averaged around 29% to 35% CPU usage. Therefore, this requirement cannot be stated as passed. However, as it is not far off, it can be said with some certainty that it would be possible to bring the CPU utilization under the 30% threshold.

5.2.3 Non-functional requirement 3 (command delay)

The non-functional requirement that is tested here is "the delay from a mouse or keyboard command given until the result of execution being shown in the *slave application window* must never exceed 3 seconds".

Two tests were performed as a part of this non-functional requirement. One where the keyboard was the focus of the test and another test where the focus was on the mouse.

WordPad was used for testing the keyboard. The text was typed at a tempo of approximately 3 chars per second for about 20 seconds. It was observed that it took longer than 3 seconds for the final character to be displayed in the *slave application window*.

Paint was used for testing the mouse. Here 20 lines were drawn over the period of 10 seconds using the Pencil tool with one down and one up event for each line. And again after drawing the last line, it took more than three seconds before it appeared.

It is important to mention that the first action actually does happen within the first three seconds. However, the commands are executed slowly and the commands queue up so the last command in the queue gets executed with a significant delay.

6 Results and discussion

6.1 Table of test results

Test results of *nice to have scenarios* are not included in the table of test results.

***Scenario* name**	**Passed/Failed**
***Use case*: Manage account**	
Create account	Passed
Login to account	Passed
Logout of account	Passed
***Use case*: Launch a specific application**	
Launch a specific application	Passed
***Use case*: Control a running application**	
Use of left and right mouse buttons, both up and down events	Passed
Keyboard control - Character keys, enter and backspace, both down and up events	Passed
***Use case*: Manage personal files in the system**	
Upload file	Passed
Download file	Passed
Use file already in the system from within an application	Passed
Get a file from a running application to the system	Passed
Non-functional requirement:	
"The *client application* must run on Windows 10"	Passed
"The *client application* must be able to run on a low-end laptop CPU with an average CPU Mark score of 4967 (PassMark, 2019) with three concurrent *client applications* running, never exceeding 30% utilization"	Failed
"The delay from a mouse or keyboard command is given until the result of execution being shown in the *slave application window* must never exceed 3 seconds"	Failed

6.2 Discussion of test results

As it can be seen in the table in section 6.1, all of the *use case* tests passed, however, two of the non-functional requirements did not. One thing to note here is that there are several more *nice to have* tests that failed which are not included in this section.

6.2.1 Discussion of failed non-functional requirement 2 (CPU utilization)

The first test that failed is a non-functional requirement that states the CPU utilization of the computer that is running the *client application* must never exceed 30%. The test showed the utilization to be hovering around 29% to 35%. However, as the team did not focus on optimization, it is conceivable that this requirement is easily achievable.

A clear opportunity for improvement of both quality and performance is afforded by implementing video streaming, as the current solution realized with the PyAutoGUI Python module is not very efficient. The screen capture would be much more efficient and smooth as a video stream as mentioned in section 3.4.7. This could be done using FFmpeg (FFmpeg contributors, 2019).

6.2.2 Discussion of failed non-functional requirement 3 (command delay)

The second test that failed is a non-functional requirement that states "the delay from a mouse or keyboard command is given until the result of execution being shown in the *slave application window* must never exceed 3 seconds". This non-functional requirement ties in with the *use case* "Control a running application" as it is the performance of the *scenarios* in this *use case* that is tested in the non-functional requirement. Here both of the *need to have scenarios* associated with the *use case* passed. However, from a user's perspective the verdict might differ, which is expressed by the failure of the non-functional requirement.

The root cause of the slow execution speed was determined to be that PyAutoGUI is rather slow. Or more specifically, PyAutoGUI cannot execute commands as fast as they are sent to the *slave module*. This causes the commands to queue up and therefore, if several commands are given in short succession, the delay exceeds 3 seconds. However, if there are no commands queued and just a single command is issued, PyAutoGUI is able to execute this command within the 3 seconds time period. Nevertheless, since the non-functional requirement states that the delay must never exceed 3 seconds, the non-functional requirement is stated to fail.

The fact that PyAutoGUI executes commands slowly is also what led to the design decision to have the mouse location update once every second, as it otherwise got very easily overwhelmed with commands to execute. Therefore to make this non-functional requirement pass, another technology would have to be used to control both the mouse and the keyboard on the *slave module*.

6.3 General discussion

It is noteworthy that all of the *need to have* parts of the *use cases* have passed their test. This means that the system now has all of the core functionality that was expected at the outset of this project. It must be brought forth that the software developed here is not yet a minimum viable product (Technopedia, n.d.) as many actions either cannot be performed, are unsecure or too slow.

"Delete account" *nice to have scenario* can serve as an example of what is still missing. Such a functionality is not necessary for demonstrating the feasibility of the envisioned system. However, the situation changes once the production ready system is considered.

Another functionality that is missing is that of streaming the audio from the *slave module* to the *slave application window*. This is something that proves difficult with the currently used virtualization platform Hyper-V as it does not support creation of virtual sound cards. It is therefore impossible to capture sound from the *slave module* as there is no virtual sound card for the sound to be played on and captured from. It would therefore be necessary to find a third party tool that can generate virtual sound cards or completely move to another virtualization platform.

An example of an action that currently happens too slow is a continuous update of the mouse location. As described in section 5.1, because the mouse location is updated only once every second, the drawing of a circle ends up as a polygon or simply just a line.

It should be noted that even though the testing has been done only with Paint and WordPad, there is no technical obstacle that would prevent the system from working with other more demanding applications.

7 Conclusion

The accomplishments of the developed system are summarized below.

The developed system allows users to run applications remotely (WordPad and Paint have been tested), although the computational efficiency of the system is not great, as described in section 6.2.1. The user is able to interact with the applications with mouse and keyboard, albeit not all of the interactions are possible and, as mentioned in the section 6.2.2, the system can exhibit quite a long delay. The system allows the user to store data and can therefore act as a central storage for all of the user's data, even though a primitive one, as elaborated in section 2.2.4.

The developed system serves as a demonstration of the production ready system that would address the issues layed out in the introduction. More work outlined in the *nice to have scenarios* (section 2.2), delimitations (section 2.4) and project future (section 8) would have to be put into the project to constitute a minimum viable product.

That being said, it is now discussed how the production ready system addresses the disadvantages of the status quo.

First, as long as a device is able to use the system, its application use is not confined by the components it contains. This is an important point worth reiterating. As long as even an old power-efficient laptop has Internet access and the envisioned system installed, the user can work with power-hungry applications such as Adobe Premiere Pro, Autodesk 3ds Max, Trimble SketchUp and similar.

Second, the system has the potential to eliminate the need to replace hardware because of lacking computational performance. The same need is, however, retained for the reasons of mechanical failure of the device or better hardware outside of the computational realm, such as better screen and battery. The system can, however, change the way how computers are used and with that, prevent the migration effort associated with acquisition of a new machine that is required nowadays.

Third, the system allows the user to run applications across different operating systems. This is important because it diminishes the importance of using a particular operating system. In addition, the system can also be augmented with features valuable for the user, such as password management and file history.

Fourth, even a device that relies on the system would be idle most of the time. The point is, however, that this device is comparatively much less powerful than an average machine sold nowadays and the powerful hardware, which is situated in data centers and runs the actual applications, can be utilized very efficiently.

Fifth, the user is bereft of the responsibility to update the application as this is something the system does in the background and the user is therefore always running the most up-to-date software.

Lastly, it is important to realize that with the setup the system implements, a user's setup is not central to his local machine. Instead, it is in the cloud, accessible from whichever machine running the system after logging in.

At its current stage, the developed system does not meet a single of its afore-

mentioned aspirations. Achieving just one of them is rather challenging and requires much more work on multiple elements of the system. Most importantly, the deficiencies discussed in section 6.2 would have to be addressed. However, the developed system represents a solid basis which can be further improved to gradually attain its initial ambition.

8 Project future

There are many aspects of the project that would have to be considered and implemented to be even a minimum viable product. These include considerations about **business model**, CIA triad (Rouse, 2014) (**security** and **availability**), **privacy** concerns, system **scalability**, **legal** matters, supporting more **applications**, **hardware provisioning** and additional **features**. As Elon Musk said (Musk, 2019), "the really hard part that requires a lot of resources is optimizing something past the initial prototype phase and bringing it into volume production."

The rest of this section examines these concerns in more detail.

8.1 Business model

If this project would ever be available commercially, it is imagined that the business model would be based on a monthly subscription fee. After all, many businesses today are going down that path as well. It seems as a good way for both the customers – for whom the barrier to start using the product is not too great – and for the business itself, which gets a continuous stream of cash that in the aggregate might bring in more profits than any other model.

An analysis would have to be performed to decide upon specific details for the subscription fee, such as:

1. Is there a single subscription type or multiple ones that differ in price and offered services?
2. Are there any limitations on usage of a subscription (or a particular type), e.g.: maximum offered computation power per month, number of devices a subscription can be used from or application packs that a subscription offers?
3. What is a reasonable price that, when scaled up, covers the cost and still provides a wide profit margin?
4. Can a freemium model or otherwise limited free version be provided to get into a compounding virality loop? (Hoffman and Chestnut, 2019)

8.2 Security and privacy

As in every modern project, **security** of the system itself and the data it stores needs to be considered. This project entails more traditional security problems

such as security of database and other data files stored in the system but also a more interesting problem of securing the transmission of video and audio feed and the user's inputs over the network so the system cannot be misused and abused. Solving this problem is part of **privacy** concerns as well as only by solving this problem can the user trust the system and know that no unauthorized person can get access to his data.

8.3 Availability

Users rely on the system to be available whenever they need so availability is a key for the commercial success of the product. Solving this problem requires the system to be stable and efficient in the long run, so the system would have to better optimized.

8.4 Legal matters

As this system functions only as a bridge to other applications from other companies and developers, the obvious legal issue are licenses for the applications accessed by users.

8.5 Hardware provisioning

The applications on *slaves* and *servermodules* in the end have to be running on some hardware. This would be provisioned by one of the providers such as AWS, Microsoft Azure, etc., as is the today's standard (Amazon Web Services, 2017). Only in the case of performance optimization on the level of hardware and cost savings would it make sense to consider administering own hardware and data centers.

8.6 Scalability

Scalability means the ability to handle potentially exponentially increasing user base and therefore load on the system while expanding the capability of the system. For the system to scale well, it needs to be properly optimized.

This is partly addressed with the system being designed using individual modules that have their concerns separated as described in 3.4. This design allows the whole system to scale out as opposed to scale up (Banks, 2014) which is a preferable strategy in the long term. However, more needs to be done in order to truly allow this kind of scalability.

8.7 Application selection

The more applications the system can support, the more users can become interested in using the system. Therefore expanding the selection of available applications is important. The goal should be to allow the users to request an

application that is not currently available and the system being able to provide it in a matter of just a few minutes. Therefore, it is inherently tied to the problem of system scalability.

8.8 Additional features

8.8.1 Automatic slave initialization

An important part of the system that is managed manually at the moment is the initialization of a *slave* running the requested application. Automation of this has a high priority and is inherently interlinked with the hardware provisioning described in section 8.5.

8.8.2 Store application configuration

In order to fulfill one of the main aspirations, the system has to be able to store individual user's application configuration and apply it to the initialized application when it is launched and provided to the user.

8.8.3 Automatic application updates

Another major advantage the system can provide to its users is always using up-to-date applications without the need to spend any time updating by themselves. The system would need to update its applications while they are not being used. At the same time, the system should still provide the possibility to run an application in a certain version in case the user prefers that version over the most up-to-date.

8.8.4 Integrated system augmentations

Additionally, the system allows for more augmentation and integration of other applications, providing even more benefits to its users.

Integrated file history

One of those integrations is a file history for each file uploaded to the system so that the user can always go back to the previous version of the file.

Integrated password management

Another integration is that of a password manager which would automatically fill-in the login details to other services.

References

Alvarez, M., 2009. *The Average American Adult Spends 8 1/2 Hours A Day Starting Into Screens*. [Online; accessed 26-August-2019]. Available at: <https://atelier.bnpparibas/en/smart-city/article/average-american-adult-spends-8-1-2-hours-day-staring-screens>.

Amazon Web Services, 2017. *Netflix on AWS*. [Online; accessed 02-December-2019]. Available at: <https://aws.amazon.com/solutions/case-studies/netflix/>.

Banks, E., 2014. *What does "scale out" vs. "scale up" mean?* [Online; accessed 02-December-2019]. Packet Pushers. Available at: <https://packetpushers.net/scale-up-vs-scale-out/>.

Beach, T. E., 2000. *Types of Computers*. [Online; accessed 09-April-2019]. Available at: <https://web.archive.org/web/20150730182332/http://www.unm.edu/~tbeach/terms/types.html>.

Business News Daily Staff, Sept. 2018. *How To Set Up And Use X11 Forwarding On Linux And Mac*. [Online; accessed 27-October-2019]. businessnewsdaily. Available at: <https://www.businessnewsdaily.com/11035-how-to-use-x11-forwarding.html>.

Cornell, D., Oct. 2007. *Cleartext vs. Plaintext vs. Ciphertext vs. Plaintext vs. Clear Text*. [Online; accessed 16-October-2019]. Denim Group. Available at: <https://www.denimgroup.com/resources/blog/2007/10/cleartext-vs-pl/>.

Dignan, L., 2019. *Top cloud providers 2019: AWS, Microsoft Azure, Google Cloud; IBM makes hybrid move; Salesforce dominates SaaS*. [Online; accessed 09-April-2019]. Available at: <https://www.zdnet.com/article/top-cloud-providers-2019-aws-microsoft-azure-google-cloud-ibm-makes-hybrid-move-salesforce-dominates-saas>.

Docker Inc, n.d.(a). *Overview of Docker Compose*. [Online; accessed 22-November-2019]. Available at: <https://docs.docker.com/compose/>.

— n.d.(b). *Docker*. [Online; accessed 21-September-2019]. Available at: <https://www.docker.com>.

Electron, 2019a. *About Electron*. [Online; accessed 30-September-2019]. Available at: <https://electronjs.org/docs/tutorial/about>.

— 2019b. *Electron Apps*. [Online; accessed 30-September-2019]. Available at: <https://electronjs.org/apps>.

— 2019c. *ipcMain | Electron*. [Online; accessed 30-September-2019]. Available at: <https://electronjs.org/docs/api/ipc-main>.

— 2019d. *ipcRenderer | Electron*. [Online; accessed 30-September-2019]. Available at: <https://electronjs.org/docs/api/ipc-renderer>.

Facebook Inc., 2019a. *React - A JavaScript library for building user interfaces*. [Online; accessed 16-December-2019]. Available at: <https://reactjs.org/>.

— 2019b. *Rendering Elements - React*. [Online; accessed 16-October-2019]. Available at: <https://reactjs.org/docs/rendering-elements.html>.

— 2019c. *State and Lifecycle - React*. [Online; accessed 16-October-2019]. Available at: <https://reactjs.org/docs/state-and-lifecycle.html>.

FFmpeg contributors, 2019. *FFmpeg - A complete, cross-platform solution to record, convert and stream audio and video*. [Online; accessed 22-November-2019]. Available at: <https://github.com/FFmpeg/FFmpeg>.

Figueiredo, R., 2019. *Electron CGI*. [Online; accessed 30-September-2019]. Available at: <https://github.com/ruidfigueiredo/electron-cgi>.

Gilbert, B., 2018. *The PlayStation 4 continues to dominate as the world's most popular gaming console*. [Online; accessed 09-April-2019]. Available at: <https://www.businessinsider.com/ps4-playstation-4-lifetime-sales-2018-1>.

GitHub, 2019. *css · GitHub Topics*. [Online; accessed 30-September-2019]. GitHub. Available at: <https://github.com/topics/css>.

GURU99, n.d.(a). *Positive Testing and Negative Testing with Examples*. [Online; accessed 18-September-2019]. Available at: <https://www.guru99.com/positive-and-negative-testing.html>.

— n.d.(b). *What is User Acceptance Testing (UAT)? with Examples*. [Online; accessed 04-December-2019]. Available at: <https://www.guru99.com/user-acceptance-testing.html>.

— 2019. *Top 21 Cloud Computing Service Provider Companies in 2019*. [Online; accessed 10-October-2019]. Available at: <https://www.guru99.com/cloud-computing-service-provider.html>.

Hintjens, P., 2019. *ØMQ - The Guide*. [Online; accessed 21-November-2019]. iMatix. Available at: <http://zguide.zeromq.org/page:all/#The-Socket-API>.

Hoffman, R. and Chestnut, B., May 2019. *The Case For Bootstrapping*. [Online; accessed 14-October-2019]. Masters of Scale. Available at: <https://mastersofscale.com/wp-content/uploads/2019/05/mos-episode-transcript-ben-chestnut-.pdf>.

Hunt, P., 2019. *React: Rethinking best practices*. [Online; accessed 30-September-2019]. Youtube. Available at: <https://www.youtube.com/watch?v=x7cQ3mrcKaY>.

Isikdogan, L., 2018. *How Video Compression Works*. [Online; accessed 22-November-2019]. Youtube. Available at: <https://youtu.be/QoZ8pccsYo4>.

LaMarco, N., 2018. *The Average Lifespan for Laptops*. [Online; accessed 09-April-2019]. Available at: <https://smallbusiness.chron.com/average-lifespan-laptops-71292.html>.

McAfee, A., 2019. *More from Less Overview — Andrew McAfee*. [Online; accessed 05-October-2019]. Available at: <https://andrewmcafee.org/more-from-less/overivew>.

McKalin, V., 2018. *Broken image icon in Google Chrome browser*. [Online; accessed 9-December-2019]. Available at: <https://www.thewindowsclub.com/broken-image-icon-google-chrome-browser>.

Microsoft, Sept. 2018. *Application Virtualization (App-V) for Windows 10 overview*. [Online; accessed 16-October-2019]. Microsoft. Available at: <https://docs.microsoft.com/en-us/windows/application-management/app-v/appv-for-windows>.

Microsoft, 2019. *TypeScript - JavaScript that scales*. [Online; accessed 16-December-2019]. Available at: <https://www.typescriptlang.org/>.

Mozilla Contributors, 2019a. *Chrome - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. [Online; accessed 03-October-2019]. Available at: <https://developer.mozilla.org/en-US/docs/Glossary/Chrome>.

— 2019b. *WebRTC API - Web APIs | MDN*. [Online; accessed 22-November-2019]. Available at: <https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API>.

Musk, E., 2019. *Starship Update*. [Online; accessed 14-October-2019]. Youtube. Available at: <https://youtu.be/sOpMrVnjYeY?t=3565>.

Nadig, S., 2019. *What Is Negative Testing And How To Write Negative Test Cases?* [Online; accessed 14-November-2019]. Software Testing Help. Available at: <https://www.softwaretestinghelp.com/what-is-negative-testing/>.

NLog, 2019. *NLog - Flexible & free open-source logging for .NET*. [Online; accessed 30-September-2019]. Available at: <https://nlog-project.org/>.

PassMark, 2019. *PassMark - Intel Core i5-6267U @ 2.90GHz*. [Online; accessed 06-September-2019]. Available at: <https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i5-6267U+%40+2.90GHz>.

Puget Systems, n.d. *Recommended Systems for Adobe Premiere Pro*. [Online; accessed 29-November-2019]. Available at: <https://www.pugetsystems.com/recommended/Recommended-Systems-for-Adobe-Premiere-Pro-143/Hardware-Recommendations>.

Red Hat, n.d. *What is Docker?* [Online; accessed 04-December-2019]. Available at: <https://opensource.com/resources/what-docker>.

Robinson, D. and Coar, K. A. L., Oct. 2004. *The Common Gateway Interface (CGI) Version 1.1*. RFC 3875. RFC Editor. Available at: <http://www.rfc-editor.org/rfc/rfc3875.txt>.

Rouse, M., n.d. *cloud storage service*. [Online; accessed 09-April-2019]. Available at: <https://searchstorage.techtarget.com/definition/cloud-storage-service>.

— 2014. *What is confidentiality, integrity, and availability (CIA triad)?* [Online; accessed 10-October-2019]. WhatIs.com. Available at: <https://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA>.

Sorhus, S., 2019. *Useful resources for creating apps with Electron - Boilerplates*. [Online; accessed 30-September-2019]. Available at: <https://github.com/sindresorhus/awesome-electron/#boilerplates>.

Sweigart, A., 2019. *Welcome to PyAutoGUI's documentation!* [Online; accessed 26-September-2019]. Available at: <https://pyautogui.readthedocs.io/en/latest/>.

Technopedia, n.d. *What is a Minimum Viable Product (MVP)? - Definition from Techopedia*. [Online; accessed 04-October-2019]. Available at: <https://www.techopedia.com/definition/27809/minimum-viable-product-mvp>.

Unreal Streaming Technologies, 2019. *Unreal Media Server FAQ | What are best practices for lowest latency live streaming?* [Online; accessed 22-November-

2019]. Available at: <http://umediaserver.net/umediaserver/faq.html#What-are-best-practices-for-lowest-latency-live-streaming>.

Vītoliņš, K., 2019. *Create a desktop app with Electron, React and C#*. [Online; accessed 30-September-2019]. Available at: <https://itnext.io/create-desktop-with-electron-react-and-c-86f9765809b7>.

Vocke, H., 2018. *The Practical Test Pyramid*. [Online; accessed 14-November-2019]. Available at: <https://martinfowler.com/articles/practical-test-pyramid.html>.

Wikimedia Commons, 2018. *File:ISO keyboard (105) QWERTY UK.svg — Wikimedia Commons, the free media repository*. [Online; accessed 9-December-2019]. Available at: <https://commons.wikimedia.org/w/index.php?title=File:ISO_keyboard_(105)_QWERTY_UK.svg&oldid=331373586>.

Wikipedia contributors, 2019a. *Feature toggle — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-November-2019]. Available at: <https://en.wikipedia.org/w/index.php?title=Feature_toggle&oldid=920378491>.

— 2019b. *Sass (stylesheet language) — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-December-2019]. Available at: <https://en.wikipedia.org/w/index.php?title=Sass_(stylesheet_language)&oldid=929083869>.

— 2019c. *The quick brown fox jumps over the lazy dog — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-November-2019]. Available at: <https://en.wikipedia.org/w/index.php?title=The_quick_brown_fox_jumps_over_the_lazy_dog&oldid=926313765>.

— 2019d. *Video coding format — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-November-2019]. Available at: <https://en.wikipedia.org/w/index.php?title=Video_coding_format&oldid=927055403>.

— 2019e. *Webpack — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-December-2019]. Available at: <https://en.wikipedia.org/w/index.php?title=Webpack&oldid=929083977>.

ZeroMQ, n.d. *ZeroMQ*. [Online; accessed 19-September-2019]. Available at: <https://zeromq.org/>.

Appendices

A Test of use cases

Test of use case: Manage account

This *use case* has 3 defined *need to have scenarios*, which are tested one by one.

Test of scenario 1 - Create account

Precondition: Precondition having launched the *client application*
Expected outcome: Get logged into the newly created account

Test steps:

1. Press "Create account"
2. Enter valid required information
3. Press "Create account and login"

Executed steps:

1. Press button "Create account"
2. Enter "test@ccfeu.com" into "Email address" field
3. Enter "password" into "Password" field
4. Press button "Create account and login"
5. Press button with gear icon
6. Observe the email currently logged in is also "test@ccfeu.com"

Observed outcome: It was observed that the *client application* was logged into the newly created account, with the email "test@ccfeu.com", by observing the email address in the settings dropdown menu

Test of scenario 2 - Login to account

Precondition: Having launched the *client application* and already having created an account with login information {Email:"admin@ccfeu.com", Password:"pass"}
Expected outcome: User is logged in

Test steps:

1. Enter required information
2. Press "Login"

Executed steps:

1. Enter "admin@ccfeu.com" into "Email address" field
2. Enter "pass" into the "Password" field
3. Press button with gear icon
4. Observe the email currently logged in is also "admin@ccfeu.com"

Observed outcome: It was observed that the email "admin@ccfeu.com" was displayed as the email currently logged into, just as expected

Test of scenario 3 - Logout of account

Precondition: Having launched the *client application* and be logged into an account
Expected outcome: The login form is shown

Test steps:

1. Click on settings menu
2. Click on "Logout" from the context menu

Executed steps:

1. Press button with gear icon
2. Press "Logout"
3. Observe that the login form is now shown

Observed outcome: It was observed that the login form was shown just as expected

Test of nice to have - Manage account:

1. Update account information
    - Currently not possible
2. Delete account
    - Currently not possible

Test of use case: Launch a specific application

This *use case* has only 1 defined *need to have scenario* which needs to be tested.

Test of scenario 1 - Launch a specific application

Precondition: Having launched the *client application* and be logged in
Expected outcome: New window is created that after initialization shows the selected application

Test steps:

1. Navigate to the list of application
2. Find and click on the desired application

Executed steps:

1. Click the "Apps" button at the top of the *client application*
2. Click the "Paint" button
3. A new window appeared from which Paint could be used

Observed outcome: A new window appeared showing the application Paint

Test of nice to have - Launch a specific application:

1. Streaming visual representation of the application as a video
    - Currently not possible

Test of use case: Control a running application

This *use case* has 2 defined *need to have scenarios*, which are tested one by one.

Test of scenario 1 - Use of left and right mouse buttons, both up and down events

Precondition: Having launched a specific application
Expected outcome: See that the mouse events were activated

Test steps:

1. Hover the mouse above the *slave application window*
2. Press down on either left or right mouse button
3. Optional: move the mouse
4. Release the mouse button to activate the up event

Executed steps:

1. The application Paint is used for this test
2. Move the mouse above the Pencil tool

3. Press left mouse button down
4. Release left mouse button
5. Observe that the tool got selected
6. Move the mouse such that it is above the "Home" tab
7. Press right mouse button down
8. Release right mouse button
9. Observe that the context menu appears

Observed outcome: It was observed that both the left and the right mouse button actions occurred.

Test of scenario 2 - Keyboard control - Character keys, enter and backspace, both down and up events

Precondition: Having launched a specific application and being in a state where typing on the keyboard produces an observable outcome
Expected outcome: See that the expected key output occurred

Test steps:

1. Press key
2. Release key

Executed steps:

1. The application Paint is used for this test
2. The Text tool is selected from the toolbar
3. Create a text field by left clicking on the canvas
4. Enter the following pieces of text to demonstrate that every character key is working
   (a) " the quick brown fox jumps over the lazy dog " (Wikipedia contributors, 2019c)
   (b) " THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG "
   (c) " 1234567890-= "
   (d) " !@#$%^&*()_+ "
   (e) " []|<>{}\;',./:"?`~"

5. Furthermore, <Enter> and <Backspace> were also tested using the Text tool by typing <Enter> to create a new line and <Backspace> to delete the new line

Observed outcome: All of the keys worked and the test can therefore be deemed a success

Test of nice to have - Control a running application:

1. Continuous mouse position update
   - This was tested by drawing a circle in Paint. It was found that a circle can be drawn. However, it appears that the mouse position is not updated that often and therefore the circle tends to look more like a polygon.
2. Scrolling
   - Using WordPad as the test application it was found that scrolling does not work
3. All remaining keyboard keys, both down and up events
   - The following keyboard commands were tested in WordPad:
     - <Ctrl>+<S>, triggered a save dialog window
     - <Page Up> worked
     - <Tab> worked
     - <F10> worked
     - <Insert> worked
     - <Delete> worked
     - <Esc> worked
4. Resize the *slave application window*
   - It was not found possible to resize the *slave application window*.
5. Changing the local cursor so it matches the one on *slave module*
   - Did not occur during testing

Test of use case: Manage personal files in the system

This *use case* has 4 defined *need to have scenarios*, which are tested one by one.

Test of scenario 1 - Upload files

Precondition: Having launched the *client application*, be logged in and having navigated to the 'Files' tab
Expected outcome: The uploaded file appears in the list of files

Test steps:

1. Press "Upload file" button
2. Select a file using the file explorer

Executed steps:

1. Click button "Upload new file"
2. Select file "test.txt" from the desktop of the local computer, using the file picker dialog window
3. Observe that the file do appear in the list of files

Observed outcome: A file with filename "test.txt" appeared in the list, just as expected

Test of scenario 2 - Download file
Precondition: Already having at least one file in the system
Expected outcome: The selected file is downloaded to "Downloads" folder on the local PC

Test steps:

1. Select a file
2. Press "Download file" button

Executed steps:

1. Click on the file with name "test.txt" in the list of files
2. Click the button "Download file"
3. Navigate to the local computer's "Downloads" folder
4. Open the file to see that the content is as expected

Observed outcome: The file appeared in the "Downloads" folder with the expected content

Test of scenario 3 - Use file already in the system from within an application
Precondition: Already having at least one file in the system and having a running application
Expected outcome: The selected file can be opened in the application

Test steps:

1. Select the file to send

2. From the *main application window*, press "Send file to an application" button
3. From the dropdown menu select an application to send the selected file to
4. Open the file from within an application in a usual way. The file is found in a designated file location.

Executed steps:

1. The application Paint is used for this test
2. Upload a file "test.png" to the system
3. Select "test.png" file and click the button "Send file to an application"
4. Click "Paint"
5. Using the *slave application window* do <Ctrl>+O
6. Navigate to "ccfeu-files" folder located in Desktop
7. Open file "test.png"
8. The file from the system can now be used

Observed outcome: It was observed that the file could be used using the *slave application window*, just as expected

Test of scenario 4 - Get a file from a running application to the system
Precondition: Having performed steps described in "executed steps" of *scenario* "Use file already in the system from within an application"
Expected outcome: The list of files is updated and changes are present

Test steps:

1. Save changes to "ccfeu-files" folder located in Desktop
2. Click to close the *slave application window*
3. When the *slave application window* is closed, the files are saved to the system

Executed steps:

1. The application Paint is used for this test
2. Modify the image using the Pencil tool
3. Press <Ctrl>+S to save the changes

4. Close the *slave application window* using the cross button in the top right corner
5. Download the file
6. Observe the changes made using the *slave application window*

Observed outcome: It was observed that the changes made using the *slave application window* were present just as expected

Test of nice to have - Manage personal files in the system:

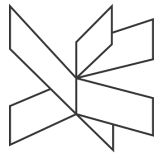
1. Rename file in the system
    - This was tested by uploading a file "test.txt" to the system. When that was done, the file was selected from the list of files, a new name "test.test" was typed in the text field and the "Rename file" button was pressed and it was observed that the renaming occurred.
2. Organize files in the system using folders
    - This is currently not possible

B Source code

Source code can be found in the accompanying .zip and on
`https://github.com/cloud-computing-for-end-users`

C Project description

See next page.



Cloud computing for end users

Bachelor project description

Kenneth Nørholm 254309
Krystof Spiller 253812

supervised by
Poul Erik Væggemose

11794 characters (not including spaces)

Software Engineering, 6th semester
May 11, 2019

Document versions:

Version	Change	Date
0.1.0	Initial draft (no sources, one subproblem is being reevaluated)	2019/04/05
0.1.1	Added sources, clarification of problem statement, delimitation section revised	2019/04/11
0.1.2	Added one subproblem about system security and delimited against it	2019/04/18
1.0.0	Time schedule section updated, small text revisions and aesthetic fixes	2019/05/11
1.0.1	Typo fixed. Date and version format changed in regards to team conventions.	2019/09/06

Contents

1 Background description

2 Definition of purpose

3 Problem statement

4 Delimitation

5 Choice of models and methods

6 Time schedule

7 Sources of information

1 Background description

If we have a closer look at the current status quo in computing for regular users we find that the user needs to have their own hardware that does the actual computation. One can find many disadvantages with such an approach.

First, the hardware needs to be exchanged every so often, on average every 5 years (Durden, 2018), for a new one because of the hardware obsolescence and therefore lacking computational performance. This will necessarily require some time to be spent selecting the new model and setting it up, as well as paying the upfront cost of the computer. Furthermore the setting up of a new computer can be a frustration with installing all of the software from the previous machine, and copying the existing data. In case of laptops it means exchanging the whole machine instead of only the parts involved in computation which is also unnecessarily wasteful.

Second, the hardware the user bought has only a limited computational potential or use case that stays the same for the rest of the hardware lifetime. This means that if this hardware has been bought for ordinary office work, one cannot expect to be able to play the latest games on it as well. On the other hand in case of gaming consoles, which is just another piece of computational hardware many people buy (Gilbert, 2018), one cannot expect to be able to do any office work. A lot of hardware is also made for a specific form factor further limiting the machine's potential. Consider for example the constrains imposed on laptop manufacturers.

Third, the hardware is tied to a certain operating system that allows to use features and applications available only on that system. Although you can run many operating systems on one machine, you will nonetheless have a problem if you want to run two applications that are each available only on a different operating system.

Fourth, the fact that the user and only the user owns and uses this hardware means that it in fact sits idle and unused most of the time (Alvarez, 2009). This strategy is wasteful, especially if we consider the relative ease of centralizing processing power, which allows for a much higher utilization (Dignan, 2019). Assume a conservative estimate that the hardware is being used for 25% of the time and stays idle for the remaining 75%. This means that the world needs four times more hardware than if the hardware would be used non-stop without being idle.

Lastly, if a user does not have the hardware with them, they cannot access their machine and use it. Data sharing services such as Dropbox, Google Drive or Microsoft OneDrive (Rouse, n.d) allow users to put their data to a cloud, making them accessible from every computer with an Internet access. As of now however, a solution that would provide the same comfort accessing a whole user's setup does not exist.

This bachelor project looks into a solution that alleviates the aforementioned disadvantages. A solution that is inspired by historical approach to computation with mainframe and client (Beach, n.d) and mimics an approach of cloud computing

services, only in this case directed at end users rather than businesses.

2 Definition of purpose

A purpose of this project is to change the status quo in computation by avoiding limited computational potential of the hardware and specific use case imposed mainly by the operating system, minimize the initial investment and additional investment of both time and money in the future associated with computation, decrease the hardware idle time and reduce environmental impact of hardware production and finally provide access to the user's environment from any computer with Internet connection.

3 Problem statement

How can a user[1] run any application(s), notwithstanding the OS, simultaneously using the same data without buying expensive hardware themselves?

Subproblems:

1. How can the user avoid buying expensive hardware?
2. How can it be made possible for a user to run any application notwithstanding the operating system?
   - How can any operating system be supported?
   - How can any application be supported?
3. How can the applications share data?
4. How can a user run multiple applications simultaneously?
5. How to make the system as responsive as is required for it to be useful?
6. How to make the system scalable?
7. How can users data be secured?
8. How can the system be secured against unintended use?

[1] In order to clarify the problem domain, three types of users are considered:
Type 1: A content creator starting out could use this system for resource intensive tasks such as video rendering and editing.
Type 2: A regular computer user who uses their computer for ordinary activities but finds that a computer is too expensive and would like to have a much cheaper option, even if this would require an Internet connection to function.
Type 3: An advanced user that is able to utilize other advantages of the system, such as being able to access their environment from any computer with Internet connection or using any application notwithstanding the operating system on which it runs.

9. How can it be ensured that the system is available for 99.99% of time?
10. How can legal ramifications for providing access to our system be avoided?

4 Delimitation

One of the subproblems that will not be considered further in this project is subproblem 7. Storing users data securely will not be considered. However, the system will store data for each user separately. The reason why other security considerations are not included into the project is that it would take time away from developing the core functionality of the system.

One more subproblem that relates to security is subproblem 8. Not only the users data need to be secured when the product comes to production, but the whole system needs to be secure and prevent any unintended use. However, for the purpose of this proof of concept application this requirement will not be taken into consideration.

Next subproblem that will not be considered is number 9 - how to ensure that the system will have an uptime of more than 99.99%. This particular subproblem only becomes important when the product is used by real users. Otherwise it only represents a burden during the development of the first product iteration. In order to achieve the uptime goal the resulting solution would have to be stable and efficient in the long term.

Another subproblem that will not be considered are the possible legal problems that arise with creating such a system - subproblem 10. The obvious legal issue are licenses for the applications accessed by the users. This issue will not be addressed as it is not necessary for the purposes of this proof of concept, it has no relation to a software engineering education and it is hard to confront without the necessary knowledge in the legal field.

Regarding the 2. subproblem, for the purposes of the first product iteration it will be limited to only support Windows 10 and an instance of Ubuntu. Furthermore, for each of the systems only two suitable freeware applications will be supported. This is due to the fact that the system is expected to be only a demo, and as such two applications is enough to demonstrate the proof of concept for the system.

Subproblem number 6 — how to make the system scalable — will only be considered and partially dealt with if the time allows as this subproblem was estimated to be the biggest in relation to how much time will be needed to solve it, as can be seen in the next section.

5 Choice of models and methods

The whole group will be responsible for all the tasks discussed in the following table:

What - subproblem	**Why** - study this problem	**Which** - outcome is expected	**Which** - methods / models / theories will be used	**What** - is the estimated workload[2]
How can the user avoid buying expensive hardware?	This is of interest as many people do not have a big amount of disposable income so when their computer eventually requires replacement it can have a big impact on their finances.	The expectation here is a proof of concept, meaning that it is not expected that people could start using this product instead of buying new computers when this project is finished. Further development would be needed to achieve that.	Utilize theory regarding distributed systems and UML[3] as a modeling tool to design and document the system.	The time for completing this problem is included in the estimated time for subproblem 2 and 3.
How can it be made possible for a user to run any application notwithstanding the operating system?	This is relevant to study as this is part of the main functionality of the project. It benefits the end user in a way that they do not need to consider what operating system the application they want to use is compatible with.	It is expected to be completed to a degree where the principle can be demonstrated. That is, as it is specified in the delimitations, this will only be demonstrated with applications running on two different operating systems.	System design will be very much the center of attention. Furthermore theory from the area of computer networking will also be very relevant.	This is expected to take approximately 275 hours.
How can the applications share data?	This is relevant in order to retain the convenience while accomplishing the first subproblem. Achieving this subproblem allows applications to access all of the users data notwithstanding the machine and operating system on which the application runs.	This is expected to be completed to a level so that it is possible to demonstrate at least some sharing of data among applications running on different operating systems.	As a basis a theory from a Computer Networks course will be used to build an architecture that allows sharing of data.	It is estimated for this to take 136 hours.

[2] Three-point estimation (Whiting, n.d) of form $E = (E_{optimistic} + 4 * E_{realistic} + E_{pessimistic})/6$ was used to calculate the estimate

[3] Unified Modeling Language

What **- subproblem**	**Why** **- study this problem**	**Which** **- outcome is expected**	**Which** **- methods / models / theories will be used**	**What** **- is the estimated workload**
How can a user run multiple applications simultaneously?	This is relevant as users need to be able to work with many applications at the same time as they are used to.	This is expected to be completed to a degree where this functionality can be demonstrated on three applications running simultaneously. It cannot be automatically assumed that the system will be able to scale to more applications at the same time.	Theory of system design and computer communication.	It is expected for this to take 102 hours.
How to make the system as responsive as is required for it to be useful?	This is important to consider as it will uncover some non-functional performance requirements for the system. It is important to find both what is the acceptable responsiveness of the system for the user and what are the ways to achieve it.	This is expected to be completed to a limited extent where the main focus will be on refresh rate and the ping time of the system.	For this subproblem mainly the theory of computer communication will be used.	Although it is expected to be achieved only to a limited degree, it is still estimated to take 267 hours.
How to make the system scalable?	This is somewhat obvious as the system is intended to be used by many people, and as such a non-scalable system will not be of any use as soon as many users start using it.	This is expected to be done to some extent but mainly through solving other problems with scalability in mind.	Mainly an automation of system deployment has to be explored and developed here. System architecture has to be reconsidered for an efficient scalable deployment.	It is estimated that this will take up around 692 hours.

6 Time schedule

Around 1000 to 1100 hours is expected to be spent from August to December. This brings up the total of hours spent to around 1250 hours.

The deadline for the project is December 12. To have a time buffer it is planned for a transition phase to be done by December 6.

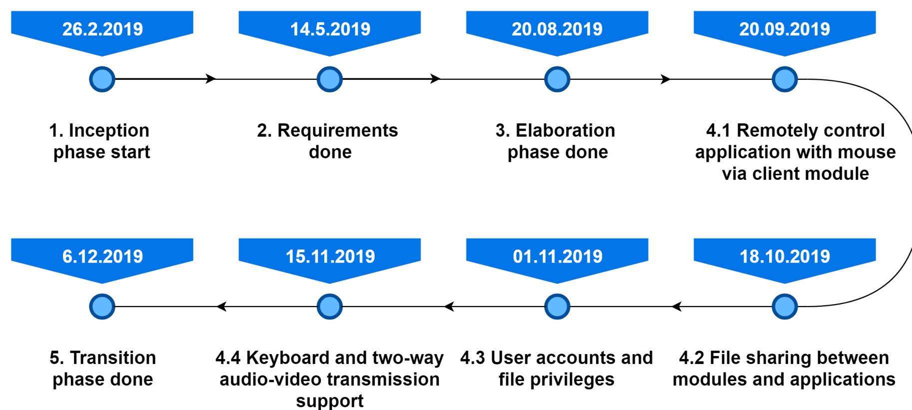


Figure 1: Timeline with milestones of the project

Following list describes some of the milestones visible in figure 1 in greater detail:

2. Project proposal, project description and SRS[4] documents are approved.

3. Conclusion of the initial analysis including the completion of a critical path analysis (Mind tools content team, n.d.).

4.1. Fundamental functional design is done and the client is able to run a remote application and control it with only mouse, meaning no other input is being transmitted.

4.2. Applications and users are able to access all of the files in the shared file storage.

4.3. Until now, a notion of a user account did not exist. This milestone implements user accounts and limits access to files that the user owns.

4.4. Support for keyboard controls is introduced. Furthermore, input from a microphone is being sent and audio is also being received by the client in addition to the video.

5. Completion of the project and process report.

[4] Software Requirement Specification

7 Sources of information

Web references:

Alvarez M., 2009. The Average American Adult Spends 8 1/2 Hours A Day Starting Into Screens. Available at: <https://atelier.bnpparibas/en/smart-city/article/average-american-adult-spends-8-1-2-hours-day-staring-screens> [Accessed 09-04-2019]

Beach, T. E., n.d. Types of Computers. Available at: <https://web.archive.org/web/20150730182332/http://www.unm.edu/~tbeach/terms/types.html> [Accessed 09-04-2019]

Dignan D., 2019. Top cloud providers 2019: AWS, Microsoft Azure, Google Cloud; IBM makes hybrid move; Salesforce dominates SaaS. Available at: <https://www.zdnet.com/article/top-cloud-providers-2019-aws-microsoft-azure-google-cloud-ibm-makes-hybrid-move-salesforce-dominates-saas/> [Accessed 09-04-2019]

Durden O., 2018. The Average Lifespan for Laptops. Available at: <https://smallbusiness.chron.com/average-lifespan-laptops-71292.html> [Accessed 09-04-2019]

Gilbert B., 2018. The PlayStation 4 continues to dominate as the world's most popular gaming console. Available at: <https://nordic.businessinsider.com/ps4-playstation-4-lifetime-sales-2018-1?r=US&IR=T> [Accessed 09-04-2019]

Mind tools content team, n.d. Critical Path Analysis and PERT Charts. Available at: <https://www.mindtools.com/pages/article/critical-path-analysis.htm> [Accessed 25-04-2019]

Rouse M., n.d. cloud storage service. Available at: <https://searchstorage.techtarget.com/definition/cloud-storage-service> [Accessed 09-04-2019]

Whiting B., n.d. Three-Point Estimating: Definition & Role in Scheduling. Available at: <https://study.com/academy/lesson/three-point-estimating-definition-role-in-scheduling.html> [Accessed 09-04-2019]

D User manual

This user manual is intended to be read by users of the system "Cloud computing for end users".

To start using the system, boot up the client application. Form here on a user account is needed.

To create an account, press the "Create account" button, then enter an email and a password for the account and press "Create account and login" button. This automatically logs you in to the newly created account. If an account already exists, simply login to that account using the email and password associated with it.

When successfully logged in to an account, there are now two different options of how to proceed. The first option is to launch applications and use these. This can be done by clicking one of the applications displayed in the list of applications. This opens a new window from where the application can be used after the initialization is done.

The second option is to go to the files tab by clicking "Files" in the navigation bar in the top. The files view shows the list of files and it is also possible to upload files to the system, download files from the system, rename them and delete them.

When files have been uploaded to the system, they can be used by the launched applications. This is done by selecting a file from the list of files, pressing the button "Send file to an application" and then clicking on the specific application that the file should be made available to.

The file is found in "ccfeu-files" folder on Desktop from where it can be opened in the application as usual.

When finished working with the file using the application, simply save the changes in the application and close the application window using the cross button in the top right corner. This saves the changes to the system.

Lastly, to logout of an account, press the gear icon on the main window and then the "Logout" button in the dropdown menu.

This concludes the simple overview of the system that is provided in this user manual.

E Authorship

Section name	Main responsible
Abstract	Krystof, Kenneth
Introduction	Krystof, Kenneth
Analysis	Krystof, Kenneth
Design - Frontend	Krystof
Design - Middleware	Kenneth
Design - Backend - File servermodule	Krystof
Design - Backend - Database servermodule	Krystof
Design - Backend - Remaining 5 sections	Kenneth
Implementation - Frontend	Krystof
Implementation - Middleware	Kenneth
Test	Kenneth
Results and discussion	Krystof, Kenneth
Conclusion	Krystof, Kenneth
Project future	Krystof