BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

APPLICATION FOR CONTROLLED ACCESS TO REMOTE DOCUMENTS FOR MICROSOFT WINDOWS

APLIKACE PRO ŘÍZENÝ PŘÍSTUP KE VZDÁLENÝM DOKUMENTŮM PRO MICROSOFT WINDOWS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ADAM FERANEC**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Abstract

This thesis aims to design, implement and test a client-side application for Microsoft Windows to ensure controlled access to remote documents. The application is programmed in C++ language, using object-oriented library MFC, WinFsp interface for virtual file system integration, and Windows API functions. The application is tested on a mock server using Python scripts and accesses the server via REST API.

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať kientskú aplikáciu pre Microsoft Windows, ktorá bude zabezpečovať prístup k vzdiaľeným dokumentom. Aplikácia je naprogramovaná v jazyku C++ s použitím objektovo orientovanej knižnice MFC, rozhrania WinFsp pre integráciu virtuálneho súborového systému a s využitím funkcí Windows API. Aplikácia serveru pristupuje cez REST API a je testovaná s využitím mock serveru a test skriptu napísaného v jazyku Python.

Keywords

Windows, application, client, C++, Python, WinFsp, MFC, filesystem, remote access, HTTP, Windows API.

Klíčová slova

Windows, aplikácia, klient, C++, Python, WinFsp, MFC, súborový systém, vzdialený prístup, HTTP, Windows API.

Reference

FERANEC, Adam. *Application for Controlled Access to Remote Documents for Microsoft Windows*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Do tohoto odstavce bude zapsán rozšířený výtah (abstrakt) práce v českém (slovenském) jazyce.

Application for Controlled Access to Remote Documents for Microsoft Windows

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of RNDr. Marek Rychlý Ph.D. The supplementary information was provided by [[**WHO**]] I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Adam Feranec
April 11, 2021

Acknowledgements

Here it is possible to express thanks to the supervisor and to the people which provided professional help (external submitter, consultant, etc.) [[**Thank for help**]].

Contents

1 Introduction **2**
1.1 Structure 2

2 Development for Microsoft Windows **3**
2.1 Development Environment 3
2.2 Windows API 4
2.3 Microsoft Foundation Class Library 9

3 Virtual File System technologies **11**
3.1 Introduction to file systems 11
3.2 Virtual Filesystems 13
3.3 Windows File System Proxy 16

4 Analysis **18**
4.1 Required technologies 18
4.2 Formalization 20
4.3 Results 20

5 Design **23**
5.1 User interface 23
5.2 Class structure 28
5.3 Data storage 29

6 Implementation **31**
6.1 User Interface 31
6.2 Internal design 31
6.3 31

7 Testing **32**
7.1 VDU server simulation 32
7.2 Modifying the application 33
7.3 Unit testing 33

8 Conclusion **34**

Bibliography **35**

Chapter 1

Introduction

Nowadays, many services allow their users to have their essential documents saved and safely backed up somewhere in the *cloud*. Be it photos, videos, or just some notes kept in a Word[1] document, today's technology allows anyone to access their files from any device. For example, imagine a typical browser-based could service. All that is required is for the user to have an internet connection and login, prove their identity to the *server*, and the application on the user's device, the *client*, takes care of the rest. After successful authentication, the user's files are available to read, download, and upload. Manually, these actions can become quite a bit of an overhead as the file count increases. What if the number of users that have access increases as well?

The answer is - controlled access to remote documents and version control, which should be present on the server's side. In this thesis's case, the basis is an internal, non-formal API description of such a server, which will be referred to as The Validated Data Storage project (VDU). As such, this thesis analyzes previously mentioned requirements of the API access and creates a client-side application for Microsoft Widnows[2] from the ground up. To allow for a better user experience, this thesis showcases an implementation of a virtual file system, present on a virtual disk, integrated right into the desktop environment of Windows. This type of integration means being able to seamlessly view and or modify a file stored in the cloud, as if it was present on a virtual disk, without the need to download or upload after each modification constantly manually.

WinFsp[27] allows for implementing a file system in the userspace for Windows and offers both lower and higher-level APIs to work with. As such, the implementation of the VDU client-side application (VDU Client) is programmed with C++. **[[CONTINUE later]]**

1.1 Structure

[[Add structure]]

[1] https://www.microsoft.com/en-ww/microsoft-365/word
[2] https://www.microsoft.com/en-us/windows

Chapter 2

Development for Microsoft Windows

Microsoft Windows is the most used desktop operating system on the market, consistently having more than a 70% market share among operating systems across many years[2]. Windows has been in development for a few decades now, and there are many ways to create applications for it. Since creating virtual file systems is not an easy task, covered in Chapter 3, this thesis uses the lower-level APIs of Windows. This chapter introduces application development for Windows using the Windows API(also known as the Win32 API), the Microsoft Foundation Class Library, and provides an overview of these technologies. The information is relevant to implement the application in Chapter 6.

2.1 Development Environment

This section focuses on all the preliminaries related to setting up a development environment for Windows desktop development.

2.1.1 Microsoft Windows Software Development Kit

The Microsoft Windows Software Development Kit (Windows SDK) is a required tool kit to develop and build applications for Windows. The Windows SDK contains all the libraries, headers, and tools required to design, implement, run, debug, and release Windows applications.

2.1.2 Operating System Version

The operating system version, also referred to as the build number of Windows, is the application's target version, and it does not always match the operating system's name. Deciding which version to target is important because of the application's backward compatibility between operating systems.[16]

This operating system version directly corresponds to the required Windows SDK version for developing applications, i.e., for Windows 10 Professional, the latest SDK is the Windows 10 SDK. It is used in this project's implementation in chapter 6. Each SDK has a list of supported operating systems. The Windows 10 SDK support goes as far as supporting Windows 7 Service Pack 1, so in theory, an application built with this SDK should run under Windows 7.[24]

Operating System	Version
Windows 10	10.0
Windows 8.1	6.3
Windows 8	6.2
Windows 7	6.1
Windows Vista	6.0

Table 2.1: Example table of versions of Windows desktop operating systems

2.1.3 Microsoft Visual Studio

The Visual Studio integrated development environment (IDE) is a program developed by Microsoft and is nearly perfect for Windows Desktop application development. It includes a code editor with state-of-the-art IntelliSense, powerful debugging tools, theme customizations, support for third-party add-ons, and even a graphic editor (Useful for designing user interface in the form of a dialog window in section 5.1.2).[23]

Visual Studio is available in three different editions: Community, Professional, Enterprise. For students, Visual Studio 2019 Community (VS19) is the best option because it is free to use under Individual Licence, allowing anyone to work and develop their own application.[22]

In Visual Studio, projects which work together are grouped under a *Solution*. A solution can contain a single project or more. Each can be built for different operating systems, with different build tools, and with different project properties.

2.1.4 Microsoft Visual C++

The Microsoft Visual C++ Toolset (MSVC), also known as the build tools, are included in Visual Studio and contain the MSVC compiler, linker, standard libraries, and headers for Windows API development. It is usually best practice to develop under the latest version of build tools. One can invoke the MSVC compiler to compile simple programs through the command line[1], but for most cases, it is preferred to let the IDE build programs while changing the options or flags, if needed, in the project's properties.[15]

2.2 Windows API

The Windows API, also often mentioned as *Win32 API*, is a massive and complex collection of headers and libraries programmed in C, containing many different macros, enums, function prototypes and can be confusing to understand at first. This section aims to give a brief overview of what is important to know about Windows API before implementing an application.

2.2.1 Integer Types

Integer data types are always capitalized. A standard signed integer is `INT`, and its size is architecture-specific. To specify how long an integer is, i.e., needing a 32-bit integer in a 64-bit architecture, it is good practice to use `INT32`. For unsigned integers, a *U* prefix is used, i.e., `UINT32`.

[1] Windows Command Prompt

By standard, a Windows Word is a 16-bit unsigned short, and its data type is `WORD`. A Double-Word is twice as long, 32-bit unsigned integer, `DWORD`. For historical reasons, Windows Word will always be guaranteed to be 16-bits long. To support the new 64-bit architecture, a Quad-Word, `QWORD` is available.

2.2.2 Pointer types

Pointer data types are defined in the form of *Pointer to X*. This is often seen directly in code or Windows API function prototypes as *P* or *LP* prefixes on data types. *P* stands for *Pointer*. *LP* stands for *Long Pointer*, a historical holdover, and for all intents and purposes, it can be considered just a regular *Pointer*. Using the standard star symbol, *, is still a valid way to signify a pointer type while programming Windows applications.

```
1 //Each of these lines is equal
2 LPDWORD pdwCount;
3 PDWORD pdwCount;
4 DWORD* pdwCount;
```

Listing 2.1: An example of declaring a pointer to a double-word

2.2.3 Code conventions

Windows uses *Hungarian Notation*[2], which adds semantical information to variable names in the form of prefixes. The information is supposed to let the programmer know the variable's intended use, data type, scope, etc., by just knowing its name without cross-referencing it. This is most often seen in Word and Double-Word variables having `w` and `dw` prefixes respectably or handles having an `h` prefix and some pointers having a `p` prefix.[25]

```
1 PDWORD pdwCount; //Pointer to a double-word variable
2 LPWSTR lpszName; //Pointer to a zero-terminated string
3 LPVOID lpBuffer; //Pointer to a buffer
4 HINTERNET hInternet; //A handle
5 LPDWORD lpcbInfo; //Pointer to a count of bytes
```

Listing 2.2: An example of hungarian notation

Similarly, many functions expect a range of values, referred to as inputs, in their calling parameters. These inputs' semantics are not always recognizable just by looking at the variable's data type. It is often generic, meaning it holds little to no information about what exactly does the function expects its input to be.

```
1 int WINAPI GetSystemMetrics(int nIndex);
```

Listing 2.3: GetSystemMetrics prototype

Listing 2.3 shows an example of an unclear expected input value `nIndex`.[29]

2.2.4 Character set

Functions, which manipulate characters are generally implemented in one of the following ways:

[2] https://web.mst.edu/~cpp/common/hungarian.html

- ANSI[3] version, signified with the suffix *A*, i.e., `InternetOpenA`
- Unicode version, signified with the suffix *W*, i.e., `InternetOpenW`
- An adaptive, generic version, with no suffix, i.e., `InternetOpen`. It is not implemented per se, rather defined as a macro, referring either to the ANSI or Unicode version, depending on the character set.

Some newer functions do not support ANSI and only have the Unicode version available.[39]

2.2.5 Strings

Strings usage ties closely to the current project's character set, either defined by a macro or set up in project settings (2.1.3). To take advantage of the Unicode character set when possible and fall back to ANSI, when it is not, it is a good practice to know about and use *portable run-time* functions and prototypes. Both prototypes and functions provide the programmer with a way to work with strings and adapt to the preferred character set automatically, recognizable by the `T`, `_T`, or `_tcs` prefixes.

```
1 char* str = "C String";
2 WCHAR* str = L"Wide string";
3 //_T is an alias of _TEXT macro
4 TCHAR* str = _T("Portable String");
```

Listing 2.4: An example of defining static strings

As such, the `_tcs` family of functions substitutes one-to-one with `wcs` and `str` family of functions. i.e., using `_tcslen` substitutes `wcslen` for Unicode character set and `strlen` for ANSI character set.[40]

2.2.6 Windows

A window is a programming construct which:

- Occupies a certain portion of the screen
- May or may not be visible at a given moment
- Knows how to draw itself
- Responds to events from the user or the operating system

By this definition, a *window* in Windows programming might not always refer to the *application window*. A button, text field, check box, or even a combo box is a window in itself. The difference is that the application window, also referred to as the *main window*, is not part of any other window of the application. The main window also often has a title bar, a minimize button, a maximize button, and other standard UI elements.

A window can have relationships with other windows. If another window creates a window, the relationship between them is *an owner/owned* relationship. If a window resides inside another window, it is called a child window. The relationship between them is *parent/child*.[42]

[3] American National Standards Institute codes https://www.ansi.org/

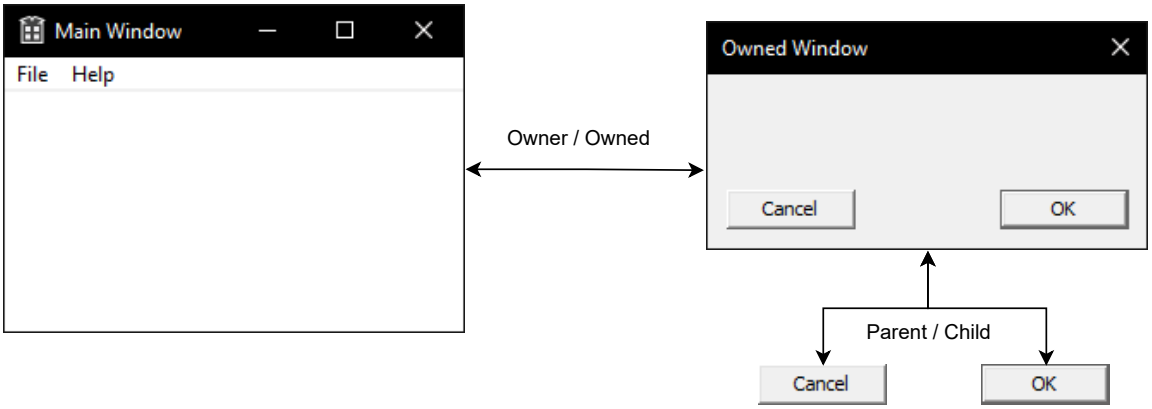

Figure 2.1: Example of owner/owned and parent/child relationships between windows.

2.2.7 Object handles

In Windows, there is no direct access to system resources like files, threads, windows, or graphic images like icons. These system resources are called objects and are unrelated to the C++ object-oriented implementation of objects. For an application to be able to access an object, it needs to obtain an object *handle*.

A *handle* is an opaque data type to access a system resource via the usage of related Windows API functions, which require an object's handle to identify the said object. The value has no real meaning outside of Windows operating system. One can imagine it as an entry of an internal Windows object table. An application can obtain a handle through various Windows API functions, depending on the object the application is trying to access, i.e., using the `CreateFile`[4] function to access a file, which returns a handle on success.[10]

Handles are kept and managed internally. Depending on the object, a single object can have either multiple handles or be limited to a single handle at a time with exclusive access.[30]

2.2.8 Function results

For functions, which return handles, it is easy to tell whether or not the function succeeded at its job. Check whether or not the returned handle is invalid. On the other side, a bunch of lower-level Windows API functions returns *NTSTATUS* as a result.

NTSTATUS is a 32-bit unsigned integer value, which is the result error code of an operation, i.e., a Windows API call. This means that, in general, the value of zero means success, and anything above zero is an error code, holding information about what operation failed. This has an exception, where the values `0 - 0x3FFFFFFF` define the success status type, and values `0x40000000 - 0x7FFFFFFF` define an information status type. This is easily checked with the `NT_SUCCESS(x)` macro, where `x` is NTSTATUS.[32][33]

2.2.9 Registry

The Windows registry is a hierarchical database containing data critical for the Windows operating system's operation, services, and applications that run on it. Data is structure is

[4] https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew

essentially in a tree format, where the nodes are called *keys*. A key can contain other keys - *subkeys* and entires of data - *values*.

Registry values have a name, type, and value. Value types are mostly standard Windows types (2.2.1) like a double-word, a zero-terminated string, or a generic binary value. There are several predefined (root) keys, each serving a different purpose either for the operating system itself, services, applications, or classes. The root keys are always open and are noted by the `HKEY_` prefix.

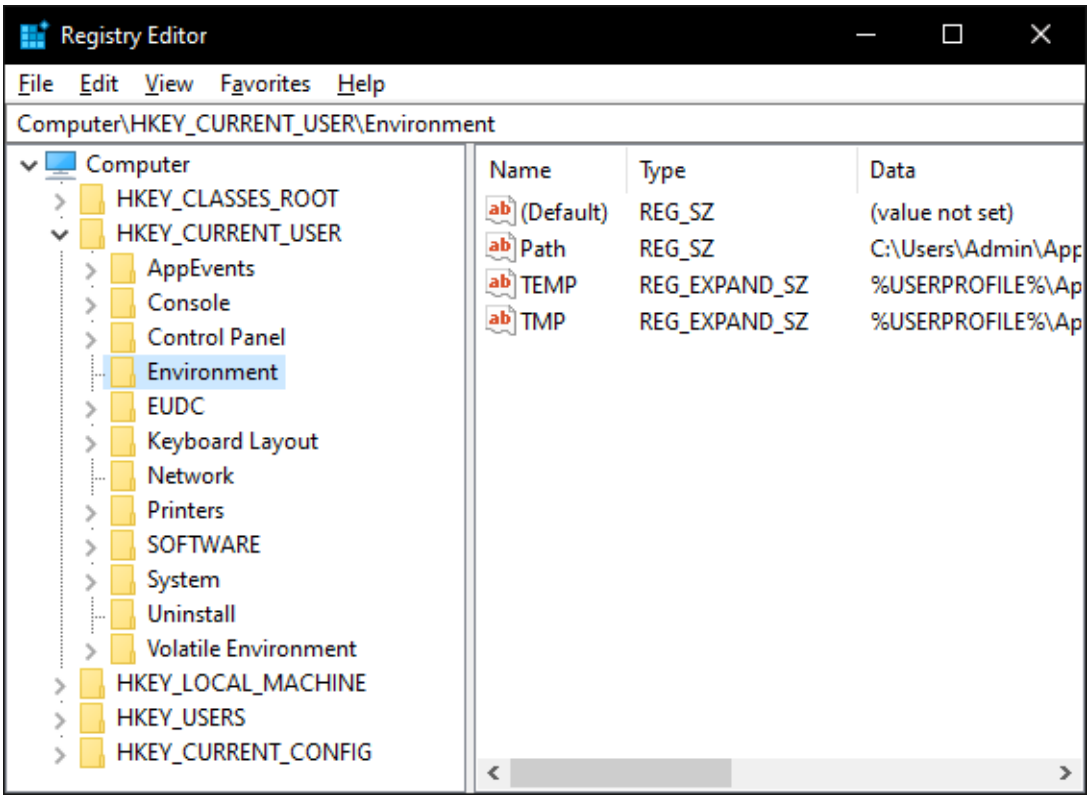

Figure 2.2: Browsing Windows registry using the Registry Editor.

To access a value of a key, one must know its path. The path is a string consisting of all the keys and subkeys, ranging from the root to the leaf key, divided by the backslash character.[35] For example, in Figure 2.2, to access a value inside the `Environment` subkey, the path would be `HKEY_CURRENT_USER\\Environment`. The Windows API provides macros for root key specification, which would allow the programmer to emit the specified root key from the path.

For an application, the registry can save user preferences, various settings, remember selected options, or track the application's usage. It is also useful to make the application run automatically upon startup, as implemented in section **[[autorun]]**.

2.2.10 Thread synchronization

There are many ways to synchronize threads in Windows. These include, but are not limited to: Events, Semaphores, Mutexes, Interlocked API, and Slim reader/writer locks (SRW Locks). As this project makes use of SRW locks, this subsection will explain only those in the following.[38]

An SRW lock is a simplified version of a semaphore, a synchronization object which is useful in controlling a shared resource between multiple threads. A semaphore has a set number of threads that are allowed to access the resource simultaneously. When a thread is done with using the resource, another thread is allowed to use it.[36] An SRW lock takes the thread's intent with the shared resource into account and is optimized for speed and performance. If a thread wants to read a resource, it can lock the resource in a *shared mode*. If a thread wants to write to a resource, it can lock the resource in the *exclusive mode*. If a resource is not locked, it can be locked in either mode.

The exclusive mode works just like a semaphore with a single allowed thread. The access is always exclusive as no other threads can simultaneously access the resource, even if some threads only want to read the resource. The shared mode allows for read-only access to the resource by multiple threads if the lock is not locked in exclusive mode.

Neither mode has a priority of acquiring the lock, there is no order or a queue of access, so if two threads want to lock an SRW lock, it is not predictable which thread will acquire the lock in different modes. An SRW lock is the size of a pointer, which means faster access and a limited amount of information stored about the lock. It is a good, simple choice for a project like this and is even sufficient to solve *„The Readers-Writers Problem*[5]. [37]

2.3 Microsoft Foundation Class Library

This section introduces the Microsoft Foundation Class Library (MFC) and aims to provide an overview of important designing functions, implementing the user interfaces.

MFC is a wide-ranged object-oriented C++ library, which abstracts and wraps much of the non-object-oriented Windows API. It is useful for designing and creating user interfaces, small or large dialog boxes, windows, implementing network services, network communication, threading, and more.[13]

2.3.1 Relations to Windows API

As mentioned in previous sections, MFC allows for much easier desktop application development by abstracting and wrapping a lot of the Windows API, originally only written in C, into the object-oriented C++ programming language.

```
1 //Windows API - Using C
2 HWND hMainWnd = CreateWindowW(...);
3 ShowWindow(hMainWnd, SW_SHOWNORMAL);
4
5 //MFC - Using C++
6 AfxGetMainWnd()->ShowWindow(SW_SHOWNORMAL);
```

Listing 2.5: Showing a window using Windows API and MFC

Listing 2.5 showcases an example of showing the main window (2.2.6) using both APIs and an instance of abstracting the window handle away in favor of using a window C++ object. Calling the `ShowWindow`[6] function directly from a window object is a lot more straightforward and convenient than handles. However, it is important to keep in mind that MFC still internally uses the Windows API. This means, if there is a need for a handle

[5] https://www.u-aizu.ac.jp/~yliu/teaching/os/lec07.html
[6] https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-showwindow

of an MFC object, there are supportive functions like `GetSafeHwnd`[7], which return the internal object handle.

2.3.2 Coding conventions

In MFC, all global static functions are marked with an `Afx`, prefix (Application Framework Extension).

2.3.3 Wrappers

2.3.4 Strings

2.3.5 Exception handling

[[Overview key MFC parts for this project]]

[7] https://docs.microsoft.com/en-us/cpp/mfc/reference/cwnd-class?view=msvc-160#getsafehwnd

Chapter 3

Virtual File System technologies

This chapter serves as an overview of available virtual file system technologies that would allow for direct integration with the Windows desktop environment. Such an instance of virtual file system implementation is a Filesystem in userspace (FUSE), "a file system in which an ordinary userspace process provides data and metadata.„[6]

This exact implementation does not exist on Windows without a kernel-mode driver[31]. Since creating a kernel-mode driver is out of this thesis's scope, the details of how this can be implemented using a third-party virtual file system software are shown in section 3.2.1.

The following sections contain the introduction to files, file systems, and an overview of available third-party virtual file system software that can be considered a valid option for this project's intent.

3.1 Introduction to file systems

The following section helps to understand what a file system is, which operations are the file system's responsibility, how it talks to the file system, how it is defined on a typical file system.

3.1.1 File

Generally, in Windows, a *file* is a unit of data in a file system. A file is stored on a storage device[1] and consists of one or multiple streams of bytes, which hold related data, and a set of attributes that describe the file and its data. The file system manages it, and any application that wants to access, read, write, or execute a file or its attributes has to interact with its respectable file system to do so. A file must follow the file systems' rules, i.e., a file must have a unique name in its directory in NTFS[2].[5]

Files in Windows are never accessed directly. Instead, applications on Windows can access a file through its handle (Section 2.2.7). When a file is opened, a handle is associated with it until the requesting process terminates or the handle is closed. Each handle is unique to each process that opens a file, and depending on which type of access to the file was requested, if one process holds a handle to a file, a second process trying to open a handle to the same file might fail.[4]

[1] i.e. Hard Drive
[2] New Technology File System

3.1.2 Filesystem

A file system is a program that describes how files are stored on a storage device. It allows applications running on the system to access, read and store files. All Windows supported file systems have the following storage components:[12]

- **Volumes**
- **Directories**
- **Files**

A Volume is where the file system resides, is the highest level of organization in a file system, and has at least one partition, a physical disk's logical division.[41] For this project's purposes, only volumes with a single partition (simple volumes) will be considered. Such volume can be called a *drive* if it is recognizable and accessible by its assigned *drive letter*. A drive letter is a single capitalized letter of the alphabet ranging from A to Z, meaning Windows only supports a maximum of 26 drives with drive letters at the same time. For simplicity, the process of assigning a volume to a drive letter while making it accessible in the system will be referred to as *mounting* the volume (The system can mount volumes to directories as well).

A directory is a hierarchical collection of files, can itself be organized into a directory, and has no limitations on the number or capacity of files that it contains. The only limit is defined by the file system itself and the capacity of the storage device.[26] It is important to remember that a directory can be referred to as a file with a special flag `FILE_BACKUP_SEMANTICS` inside the Windows API.

A file (3.1.1) is the related data, and it can be organized into a directory or reside directly in the root of a volume.

3.1.3 File path formats

Windows uses the standard, traditional DOS[3] path format, which consists of the following components:

- *A volume or drive letter* - It is expected to be followed by the volume separator character[4]
- *A directory name* - The directory separator character[5] separates subdirectories within the hierarchy
- *An optional filename* - The directory separator character separates the file path and the filename

If all three components are present, the path is called an *absolute* path. If no volume is specified and the path begins with the directory separator character, it is relative to the current drive's root. Otherwise, it is relative to the current directory.[34]

[3] Disk Operating System
[4] The colon character (:)
[5] The backslash character (\)

Table 3.1: Examples of valid file paths

Path	Description
C:\dir\test.pdf	Absolute path from the root of drive *C*
\dir\test.pdf	Relative path from the root of current drive
test.pdf	Relative path from the current directory

3.2 Virtual Filesystems

A virtual file system is an abstraction of a regular file system - any information, any data, can be organized and presented as a file system. It does not require a storage device to reside on, as it can use one of the existing ones and reside and extend upon it. It can set its own rules on volume, directory, and file management and enforce them. A virtual file system's power also comes from integrating closely with the Windows operating system - hooking into the system's internal file operations and handling them in its own way.

To achieve this, this project uses a third-party virtual file system software, which exposes a virtual file system API. In general, such an API usually allows to create a virtual file system by providing the programmer with a list of file operation functions that he must implement. These usually consist of functions that handle creating files, deleting files, reading files, etc. Once these functions are implemented, the user-mode library used to implement them provides them to the kernel-mode driver, allowing this new file system to be recognized by Windows. The process of calling implemented file operations works in reverse order. For example, when opening a file, the Windows I/O[6] subsystem, which runs in kernel mode, forwards this information to the file system driver, invoking user-implemented functions request and handle the file operation (open the file).[7]

This means that a virtual file system must implement all the Windows operating system's important file operations to be functional. Upon implementation, a file system can even be shown directly in the Windows Explorer and be accessible to all running programs if the virtual drive of the file system is mounted. This is usually done internally within the third-party API. The following subsection covers some of the popular options of virtual file system software.

3.2.1 Virtual file system software

In general, there two ways a virtual file system software can implement a virtual file system API:

- Native API
- FUSE Compatibile API

A *Native API* aims to be as close to the intended system it interfaces with as possible, without potentially harmful compromises at the cost of cross-platform compatibility or other factors unrelated to the system. This type can potentially be lower-level than FUSE API and must be well documented by its provider to be usable. As noted by its name, tying closely to a single system means the API focuses on working on the intended system as seamlessly as possible. For windows specifically, native API requires two components, a kernel-mode driver and a user-mode library which interacts with it.

[6] Input\Output

- *Pros*: Good optimization, coding constructs similar to the targeted system, all features of the targeted system
- *Cons*: Little to no cross-platform compatibility, lower-level API requires deeper knowledge of the targeted system, a steeper learning curve

The *FUSE Compatible API* is meant to be compatible with FUSE, a high-level API originally only for Linux. Compatibility with FUSE allows for cross-platform compatibility with little to no changes to the virtual file system's implementation. For Windows specifically, the implementation of file systems is vastly different from Linux. The usage of a FUSE compatible API comes with its compromises, such as lower performance, or excluding some of the features only present in Windows, i.e., volume labels.[28][6]

- *Pros*: Cross-platform compatibility, easier development with higher-level API, well-documented API
- *Cons*: Lack of Windows-specific features, restricted by POSIX standards

For this thesis, it is much preferable to choose an API software option that includes a native API, as cross-platform compatibility is not a requirement. Thus, there is no need for any restrictions. Additionally, being able to use Windows-specific file system-related features is a step towards a better user experience. An open-source license of the software would also be preferred.

3.2.2 Dokany

Dokany is one of the oldest yet still fully functional pieces of virtual file system software. It was created in 2007, and while undergoing a switching of its developers, it is still being developed today.

- *Supported API types*: Native, FUSE wrapper
- *Supported languages*: C (default), Java, Delphi, DotNet, Ruby
- *Supported architectures*: x86, x64, ARM, ARM64
- *Supported desktop operating systems*: Windows 7 SP1 / 8 / 8.1 / 10
- *Open-source*: Yes
- *Provides a driver*: Yes

In conclusion, Dokany is a well-supported, stable piece of software, nearly an ideal choice for projects that pay excessive attention to software stability and compatibility while creating a file system in various, even higher-level programming languages, rather than just the low-level C.[7][3]

3.2.3 VFSForGit

Virtual File System for Git is software developed by Microsoft to enable Git[7] at a high level, enterprise scale. VFSForGit virtualizes a Git repository into a virtual file system. This is a form of integrating the files, which are not physically present on the user's computer, rather still being present on the Git repository while being displayed. The user can download the contents of the files on request via the application's user interface.

- *Supported API types*: Native GVFS Protocol[8]
- *Supported languages*: Git commands
- *Supported architectures*: x64
- *Supported desktop operating systems*: Windows 10 version 1607, or later
- *Open-source*: Yes
- *Provides a driver*: Yes

VFSForGit is a virtual file system software aimed towards usage with Git repositories, especially at larger scales. It doesn't provide many languages or options for architectures and only supports newer versions of Windows 10. With those restrictions in mind, it is still being supported and is a useful tool for accessing Git repositories in the Windows environment.[8][21]

3.2.4 WinFsp

Windows File System Proxy is a performant, stable collection of software components, which allows for implementing a virtual file system using one of its supported API layers. The focus of WinFsp is on high compatibility with NTFS, the default file system of Windows. This allows for smooth integration with the Windows environment and virtual file systems, which use or extend NTFS.

- *Supported API types*: Native, FUSE compatibility layer
- *Supported languages*: C, C++, DotNet
- *Supported architectures*: x86, x64, ARM, ARM64
- *Supported desktop operating systems*: Windows 7 and above
- *Open-source*: Yes
- *Provides a driver*: Yes

WinFsp is a great option for any virtual file system implementation, running only on Windows. Whether it is one of the older versions of the operating system, or the newer one, WinFsp provides continuous support and compatibility with those systems while keeping the officially supported languages of its API layers both lower and higher level, thanks to the inclusion of C++ and DotNet. It is a great choice for any project starting from scratch.[9]

[7] https://git-scm.com/
[8] https://github.com/microsoft/VFSForGit/blob/master/Protocol.md

3.3 Windows File System Proxy

The third-party file system software of choice for this project is Windows File System Proxy (WinFsp). VFSforGit could not be used because of its limitations and focus on Git repositories since the VDU project does not expose any Git repositories. This is further mentioned during analysis in chapter 4. Dokany was a great option, stable, supported, and similar with compatibility layers to WinFsp. On the other hand, WinFsp has native API support for C++, which allows for cleaner and easier to understand code, and offers much better performance and optimization than Dokany. Various file system operation tests that compare versions of WinFsp against Dokany and NTFS prove this. These charts are displayed in figures 3.1 and 3.2.

Figure 3.1: File comparison tests of WinFsp and Dokany. Source:[9]

Figure 3.2: Read and write comparison tests of WinFsp and Dokany. Source:[9]

3.3.1 Introduction

WinFsp is [[**Introduce**]]

Chapter 4

Analysis

This chapter will tackle the first step of creating an application, the analysis, and steps taken during an analysis of provided documentation. It will introduce required technologies to understand and handle them. Results of the analysis are present at the end of this chapter.
[[**Include google docs pdf of requirements?**]]

4.1 Required technologies

In this project, the provided documentation was received as a non-formal description of the server's REST API the client is supposed to be accessing. As this documentation is written in plain text, it requires proper analysis, understanding of the underlying technologies, and formalization into a better format. This section serves as an overview of these technologies.

4.1.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP)
[14]

4.1.2 Representational State Transfer API

The Representational State Transfer (REST) represents an architectural style for distributed multimedia systems. It's a programming style for developing RESTful web services, which allows the developer to take advantage of an existing protocol, HTTP. It conforms at the very least to the most basic REST constraints, as defined by its creator *Roy Thomas Fielding*:

- *Client-Server* - Separates the client's side and the server's side
- *Stateless* - Each request must contain all necessary information necessary to understand the request
- *Cache* - Requests must be labeled as cacheable or non-cacheable. Improves network efficiency if it is available

These constraints are further extended specifically to address web services:

- *Uniform Interface* - Increases scalability at the cost of effectivity, uses a single standardized form

- *Layered System* - Architecture composed of hierarchical layers, each component can not access other components beyond the immediate layer with which it is interacting
- *Code-On-Demand* - Allows client functionality to be extended by downloading and executing code in the form of applets or scripts to improve system extensibility

The key abstraction of information in REST is a *resource*. A resource can be anything that can be named and might be a potential target of a request, e.g., a document, an image, a data file. Interactions, i.e., manipulating a resource, happen between two parties, as noted by the first constraint, the client and server. If a client requests an operation with a resource, it sends an HTTP request, where the HTTP method is the type of operation, and the HTTP object path is the path to the resource. After processing the request, the server can inform the client about the operation's state via the HTTP status code in the HTTP response.[18]

Method	**Path**	**Description**
GET	/users	Get all users
POST	/users/john	Update an user
DELETE	/users/john	Delete an user
PUT	/users	Add an user

Table 4.1: Examples of REST API requests

4.1.3 OpenAPI

The OpenAPI Specification is an API description format for REST APIs. An entire API can be described with just a single file of the OpenAPI format, which supports file formats of either YAML[1] or JSON[2]. OpenAPI describes:

- Available endpoints and operations on each endpoint
- Operation parameters, and input/output for each operation
- Authentication methods
- Contact information, terms of use, other information

The OpenAPI format is easily readable by both machines and humans. Many third or first-party services provide a way to visualize the API in a graphical format, i.e., the Swagger Editor[3].[20]

[1] A recursive acronym for "YAML Ain't Markup Language„
[2] JavaScript Object Notation
[3] https://editor.swagger.io/

```yaml
#A simple documentation of a /ping endpoint
openapi: 3.0.0
info:
  version: '1.0'
  title: An amazing API
  description: A formal description
servers: #Server URL for testing
  - url: 'https://localhost:4443'
paths: #Endpoint descriptions
  /ping: #Endpoint path
    get: #Method
      parameters: [] #Call parameters
      description: To test a connection.
      responses: #Possible responses
        '204':
          description: Ping success!
```

Listing 4.1: An example of an OpenAPI file in the YAML format

[[More about openapi, like, basic structure? Probably not necessary tho.]]

4.2 Formalization

Considering the provided documentation, formalization means creating an OpenAPI specification based on the documentation's plain text version. A formalized specification allows for better readability, understanding, development, and testing on the developer's side. The formalized specification's concrete usage is covered in chapter 6, implementing the client, and 7 for implementing and testing a mock server.

4.2.1 Creating the specification

Formalizing the provided documentation consists of reading and understanding all the API endpoints and their access or usage restrictions and manually creating an entry for each in an OpenAPI file. Each entry has its own possible status codes, headers, and content, which endpoint could return. For this project, I used the Swagger Editor, which allowed me to document the VDU API more comfortably. The editor's great advantage is that it can render the OpenAPI specification file in an HTML[4] 5 format, as shown in Figure 4.1.

I analyzed and noted all the API access requirements, each method, its parameters, return values, and how they tie to each other from the formal, well-specified API documentation. Afterward, I discussed this information further with my supervisor, which allowed me to understand better how the API works and how it should interact.

4.3 Results

This section will summarize the result of the VDU API documentation analysis. These results are used to guide the design of the application in Chapter 5.

[4]Hyper-Text Markup Language

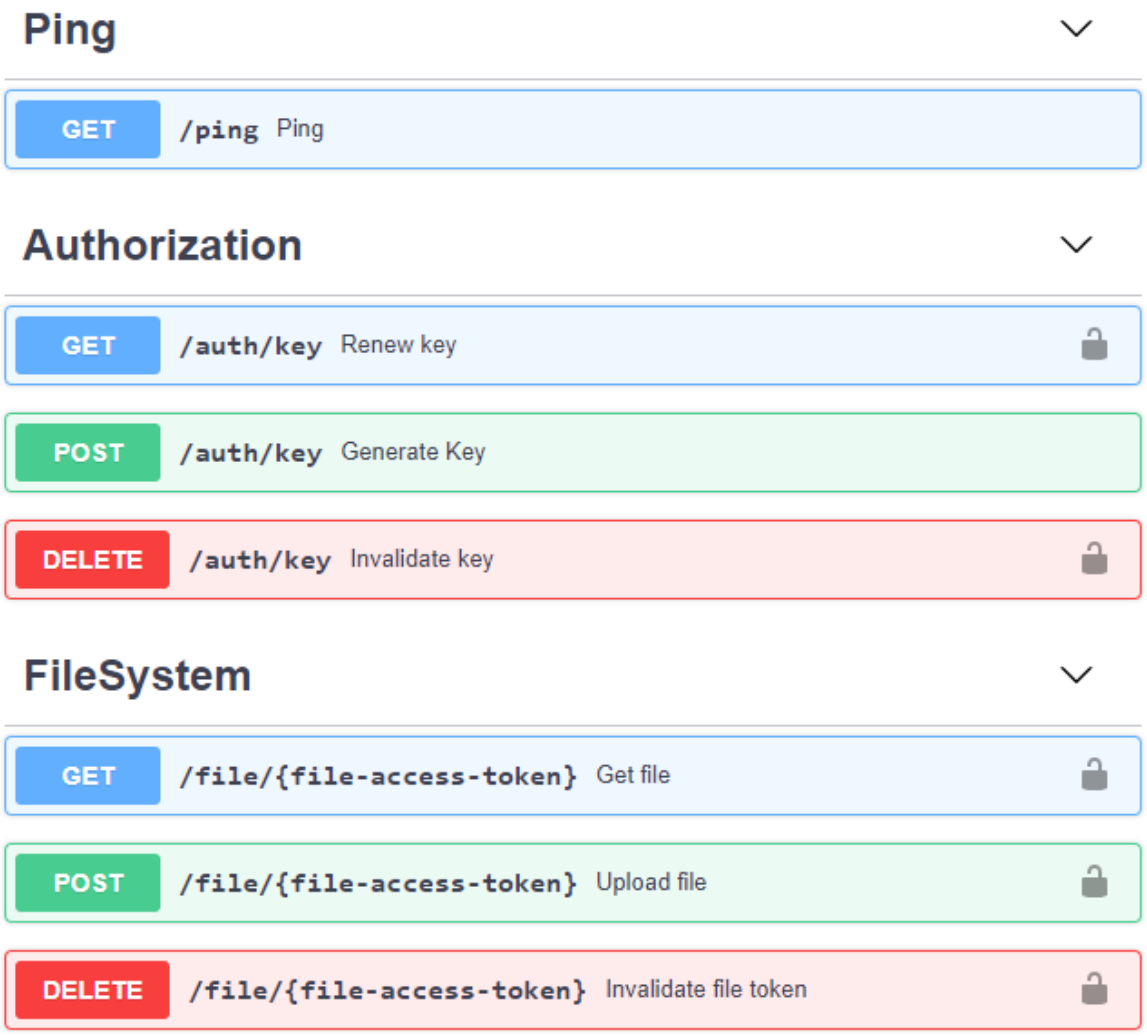

Figure 4.1: VDU REST OpenAPI 3.0 summary, rendered using the Swagger Editor, authentication requirement is signified by the lock icon.

4.3.1 Endpoints

This subsection lists all available VDU API endpoints, as shown in Figure 4.1, and provides an overview of their usages. Some endpoints require an *X-Api-Key* (API key) for successful access.

- *GET /ping* - Tests a connection to the server
- *POST /auth/key* - Authenticates an user by name or email. Client secret can be included in the content if necessary. Returns API key on success.
- *GET /auth/key* - Returns a new API key with new expiration time, refreshes session
- *DELETE /auth/key* - Invalidates API key
- *GET /file/{file-access-token}* - Returns file contents, additional file information is in the response headers

- *POST /file/{file-access-token}* - Uploads file contents, additional file information has to be in the request headers
- *DELETE /file/{file-access-token}* - Invalidates a file token, does NOT delete the file from VDU system

4.3.2 Access

Both client and server must use a secure TLS[5] channel to access the VDU API, while the server must have a valid server-side TLS certificate. This implies the usage of HTTPS protocol to access the API. The client-side certificate is optional and allows the user to omit the client secret from the authentication endpoint.

The authentication is done using an API key, which is first obtained from the `POST /auth/key` endpoint. An API key has its expiration date, which a client must respect, and the key has to be refreshed using the `GET /auth/key` endpoint if it is about to expire. The client can prematurely invalidate the API key with the `DELETE /auth/key` endpoint.

File tokens, seen as the path parameter *{file-access-token}* in the `/file/` endpoint, are generated from the proprietary VDU web user interface. Each token represents a single file, has an expiration date, and can be prematurely invalidated using the `DELETE /file/` endpoint. This file can be modified using the `POST /file/` endpoint, which includes modifying its content and file name. A file can be read-only, meaning that the server will deny any modification request.

4.3.3 Version control

The file version control system is handled on the VDU server. The VDU client does not manage or control the version of a file. This version is noted as an `ETag` header, which can be any string. The VDU server can change this tag upon successful file upload on the server's side, which could lead to invalidation of the user's file token directly after the upload. The client has to adapt to the server-side version, which it receives via a response from the `/file/` endpoint, and must not propose its own.

[5] Transport Layer Security

Chapter 5

Design

This chapter aims to design the application based on knowledge from previous chapters. The designing process consists of two main parts - the visual and the inner application design part. The visual part takes care of the user interface, user experience, all the windows, menus, buttons, and overall feeling of the application for the user. The inner application part revolves around the class structure and relationships, chooses coding practices and various internal options for designing the application, which is implemented in chapter 6.

5.1 User interface

The user interface includes all the application's visual elements and all the other elements a user can interact with while using the application. Based on the VDU API documentation analysis of requirements, I listed all important actions and ways users could interact with the VDU client. These key functionalities can be referred to as *user actions*:

- Test a connection to the server
- Login/out of the system
- Change the virtual drive letter
- Input a file token to access a file
- Read and modify accessed files, concerning version control
- Invalidate a file token of a file

The following subsections explain how the application's user interface was thought out, designed, and my thoughts behind those decisions.

5.1.1 Integration into Windows environment

Note that not all of these actions have to be included in the application's user interface. For example, the user can read and modify files via some other application present on the system, completely unrelated to the VDU client. This fact has lead me to an important realization. Windows Explorer, a handy tool for browsing files in Windows, has an amazing user interface that can display files provided by a file system - such as a virtual file system

of the VDU client. Using Windows File Explorer to provide the functionality of file access-related actions is a very user-friendly, intuitive way of integrating with the Windows desktop environment.

There is one more action that the Windows File Explorer is capable of - deleting files. Note that the VDU API does not provide an endpoint for file deletion per se. Rather, an endpoint for invalidating a file token. This means, deleting files downloaded from the VDU server in the virtual file system becomes confusing to use for the user as a file would be deleted only locally, and this action would not invalidate the token. This has lead me to an unusual design decision. Re-purposing the file deletion feature to invalidate the file's token and only delete the local file if the server allows so. Thus, the number of action which the VDU client has to implement in its own user interface is reduced even more.

In conclusion, by using built-in features of existing applications in the Windows environment, specifically the Windows File Explorer, I was able to simplify the VDU client's user interface while keeping its functionality unchanged. Details on how exactly this is implemented and why it is not limited to Windows File Explorer only are explained in chapter 6.

5.1.2 Dialog design

After narrowing down all user actions, as shown in the use case diagram in Figure 5.1, the next step is to design the actual VDU client interface.

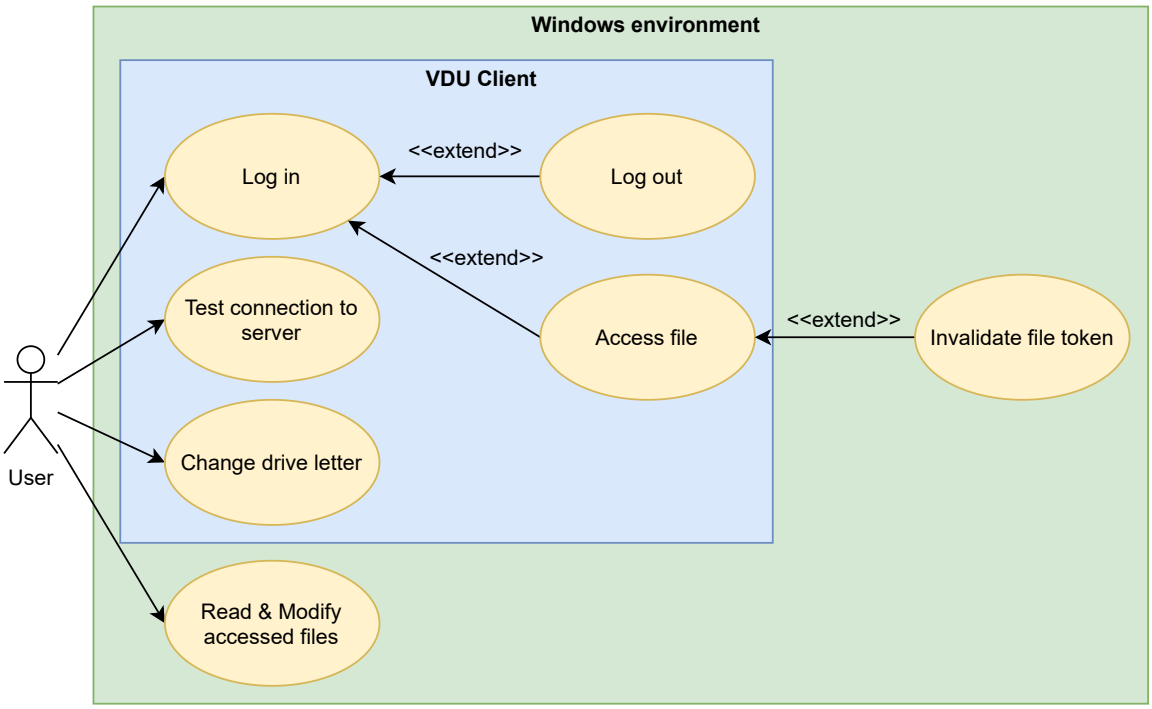

Figure 5.1: Use case diagram from the view point of an user of the VDU Client system and the Windows environment it resides in.

Considering the low amount of functionality required, I decided to go with a simple extended dialog window design(dialog). The dialog, seen in Figure 5.2, is separated into three sections:

- *Connection* - Client-Server functionality
- *File System* - Local virtual file system functionality
- *Status* - Information about the state of the application

The idea behind division was to improve visual clarity while keeping the interface compact and easy to navigate.

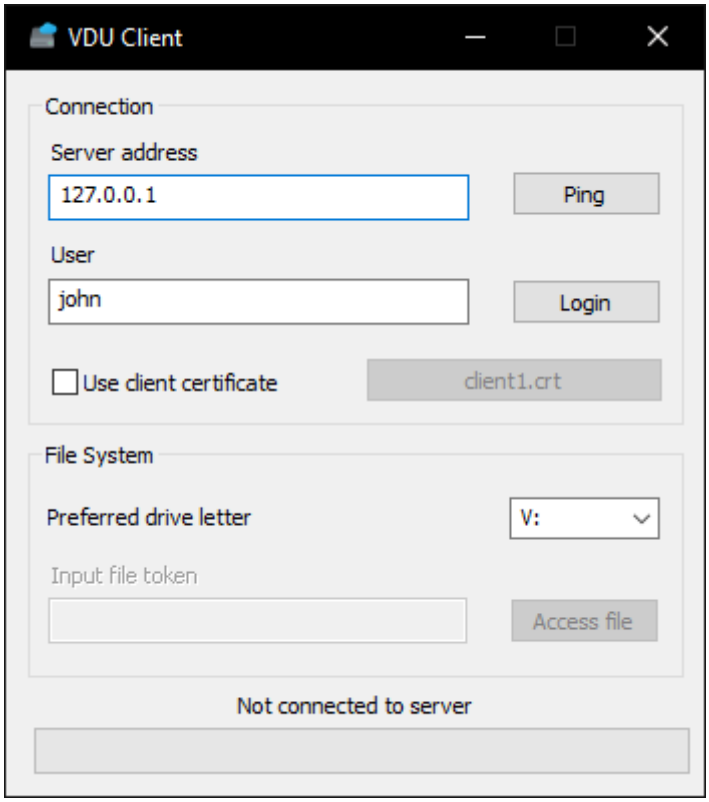

Figure 5.2: User interface of the VDU Client application.

Connection section

This section contains information about the server address, user name, and, if required, a path to a client certificate (client secret) to include the login information. Connection to the server can be tested using the *Ping* button. The *Login* button allows the user to authenticate himself to the server and changes to a *Logout* button upon successfully logging in. Logging in enables all authentication restricted functionality in the following sections and the entire application.

File system section

The *Access file* button attempts to download and launch a file from the VDU server, given a token from the input field. The user is also given an option to choose a preferred drive letter for the virtual drive, which the VDU virtual file system will control.

Status section

Containing only a progress bar and a status text, this section informs the user about the application's state. This information includes download progress - visible on the progress bar, connection status, number of accessed files, and currently logged-in user.

5.1.3 Dialog tray

When the VDU application is running, implicitly, so is the dialog window. This is true even if a user is not using the dialog window actively. In a simple use case scenario, the dialog window is only required to log in and access a file. Afterward, it theoretically does not have to be cluttering so much space on the screen and the taskbar.

This inspired me to design a beneficial addition to the dialog - a tray icon. This icon resides in the tray area of the Windows user interface. The idea is simple, upon closing or minimizing the dialog, the application would stay running in the background, signified by the icon being present. The user can restore the dialog window by simply clicking on the VDU Client icon, displayed in Figure 5.3, in the tray area.

Figure 5.3: Cloud Storage Icon, used as the main icon for VDU client application, Source:[11].

Furthermore, removing the ability to close and exit the application from the dialog window has moved this responsibility to the tray icon. I designed a simple tray menu to fix this problem, depicted in Figure 5.4, which becomes active after right-clicking the icon. The *Exit* option in this menu, shown in the listing, is how the user is supposed to quit using the application.

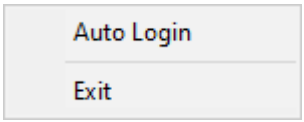

Figure 5.4: Tray menu of the VDU Client application.

Windows allows applications that put icons in the tray area to display a small text message while hovering the icon - a *tray tip*. I utilized this tray tip to display a compact, text version of the data from the status section 5.1.2 of the dialog, as displayed in Figure 5.5.

Figure 5.5: Example of a tray tip, displaying the state of the application.

5.1.4 Responsiveness and notifications

A good application needs to be responsive and notify the user about important events. Every action directly taken by the user should have a visual response. Given the specifications, the VDU client application will communicate with a server over a network connection. Whether the server resides in a local network or on the internet, it is safe to assume that the application will not finish an action instantaneously. This means a good, responsive application needs to notify the user via the user interface using two types of responses:

- *Instant* - Direct visual response to an action, proving to the user that the application is working on the user's request
- *Delayed* - A second, more detailed response, once enough information is gathered from the server

The application must keep being responsive while handling responses to all actions. How this is implemented is explained in section [[**Implementation**]].

For the VDU client dialog window, an instant response consists of enabling or disabling the related child windows[1]. For example, an instant response to clicking the *Ping* button, as displayed on the user interface in Figure 5.2, would disable the button, making it non-clickable. This button would be then enabled once again along with the follow-up - delayed response. The delayed response consists of either creating a message box or displaying a Windows notification.

Message box

A message box in the VDU client is a simple window, often created as an *owned window* relative to the main window. It is used as either an instant or a delayed response to important actions caused directly by the user, i.e., trying to test a connection to an incorrect server will result in a message box depicted in Figure 5.6.

Figure 5.6: A message box window, displayed after the application fails to connect to a server.

Windows notifications

For less important and rather informative actions, a Windows notification shows up as a response. Such a notification, displayed in Figure 5.7, appears, for example, after a successful download of a newly accessed file, along with an automatic startup of the assigned

[1] e.g. buttons, check-boxes, combo-boxes

application to the file type. This lets the user know it was the VDU client which caused the application to open.

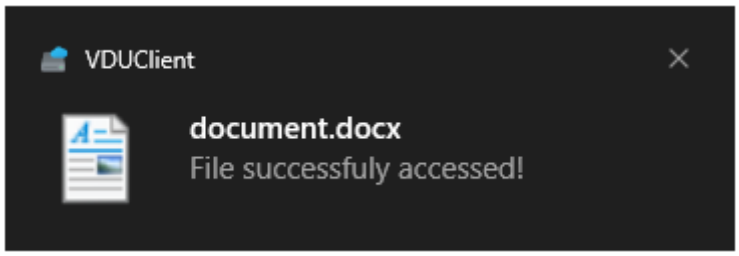

Figure 5.7: A windows notification, displayed after a successfully obtaining access to a file in the bottom right corner of the screen.

5.2 Class structure

Choosing the right design of the VDU client's internal components is key to creating a good, reliable, and scalable application. As depicted in the Unified Modeling Language (UML) class diagram in Figure 5.8, the inspiration for the concrete class structure and design was the *Single Responsibility Principle* (SRP).

According to [1], SRP is a useful design principle to keep in mind while designing object-oriented classes or modules. This is especially useful for applications, which aim to be reliable, easy to maintain, and scalable. SRP states, for example, that a single class should only have a *single reason to change*. This shifts all the responsibilities related to its purpose and allows it to be developed independently from others. Following SRP is a good way to keep software and its code clean and easy to understand. This has led me to design the following classes for the application:

- *VDUClient* - Main class of the application
- *CVDUClientDlg* - Instantiates and handles the dialog window
- *CVDUConnection* - Provides a communication layer with the server
- *CVDUSession* - Provides a session functionality for authentication purposes
- *CVDUFile* - Represents the structure and data of an accessed file from the VDU server
- *CVDUFileSystem* - Implements the virtual file system
- *CVDUFileSystemService* - Provides functionality to interact with the virtual file system

Each class has a name that implies its purpose and responsibility. All classes rely on each other to exist and can be instantiated for the VDU client to function, except the dialog handling class, `CVDUClientDlg`, which is not required to be instantiated when the application's *testing mode* is enabled. The testing mode is used while performing automatic tests on the application in chapter 7.

5.3 Data storage

There are many ways to go about storing data for an application. For the VDU client, there are two types of data, which need to be stored:

- *Files* - The actual downloaded files from the VDU server, stored using the host file system
- *Settings* - The configuration of the application, stored using Windows registry

5.3.1 Host file system

For files, it is highly advantageous to store them using the host file system over all other options available. The VDU client stores downloaded files in a temporary folder on the main drive, using the host file system. This folder exists per user, meaning each local user of Windows has their own temporary folder tied to their Windows account. This prevents unintended file sharing if multiple users are using the application on the same system. This folder is emptied each time the application starts and has a randomly generated name upon each creation. If the system crashes, all unsaved files will stay in this folder, available for potential data recovery. [[**Detailed temp?**]]

Motivation

Creating a virtual file system gives freedom to portray any data as files. This fact has inspired me to attempt to store the files using just the systems Random-Access Memory (RAM) of the application. While the speed and accessibility of RAM might make this idea seem plausible, in reality, it would not work well enough for a couple of reasons:

- *Space limitations* - Large files might not have enough space
- *Inactive RAM usage* - Files not actively in use might prevent other applications from using that space
- *No file recovery* - If the system crashes, there is no way to recover unsaved work with the files

5.3.2 Windows registry

The application's configuration does not require much space, and as such, the idea of using a custom database or storing it as a file on the host file system seems quite far-fetched. One can simplify an application's configuration settings into a simple pair `key:value`, where the *key* is a unique identifier of data type *string*, and *value* can be any supported data type on the system. The Windows registry allows applications to store information in this exact format, making it an ideal choice of data storage for the VDU client's configuration settings. The Table 5.1 lists out the concrete settings stored inside the registry by the application.

Name	Data type	Description
AutoLogin	double-word	AutoLogin feature state
ClientCertPath	string	Path to client secret file
LastServerAddress	string	Last entered server address
LastUserName	string	Last entered user name
PreferredDriveLetter	string	Selected preferred drive letter
UseCertToLogin	double-word	Whether or not to use client secret
WorkDir	string	Current directory used to store VDU files

Table 5.1: The concrete settings stored by VDU Client using the Windows registry.

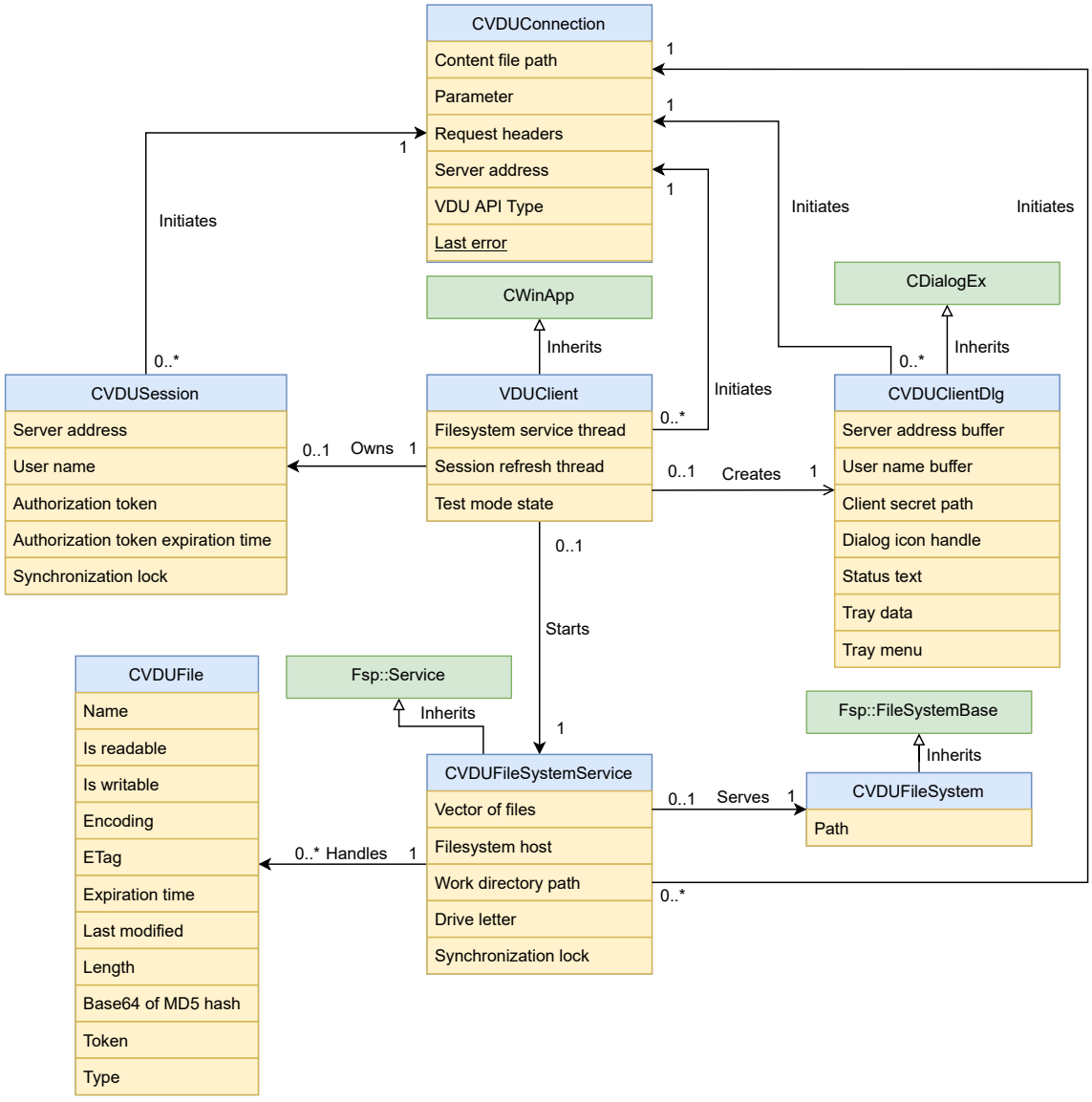

Figure 5.8: UML class diagram of the VDU Client application. [[**REDO IN THE PROGRAM**]]

Chapter 6

Implementation

This chapter covers the exact internal implementation of key elements of the VDU client, as described in previous chapter 5, which covered the overall design of application. The entire implementation of the application itself was done using Visual Studio Community 2019, an awesome IDE[1] developed by Microsoft, which was further described in section 2.1.3.

6.1 User Interface

The user interface was

6.2 Internal design

6.2.1 Thread synchronization

6.3

[1] Integrated Development Environment

Chapter 7

Testing

Application testing is a key development process used to assert and verify the application's correct functionality and the presence of its required properties. In the case of the VDU Client, the requirement is to enable the application to be tested using automated tests. Automating the testing of a Windows GUI[1] application that appears to be entirely controlled by the user's input can prove to be challenging at first. With good knowledge of the Windows API, visualizations from analysis in chapter 4 and a scripting language, I was able to simulate a VDU server and create an expandable testing script, that made implementing and running automated tests easier and scalable.

7.1 VDU server simulation

Due to the network connection requirements of the client-server type of applications, it is difficult to provably test the functionality and properties of the client without the presence of the server. Additionally, flooding a production server with automated testing might lower the quality of service for real users. This is the motivation behind creating a simulated VDU server, which would only be present locally and be able to simulate fulfilling the requests of the client with valid responses and comes with several advantages over the real VDU server:

- *Independence* - Even if the VDU server stops working, testing can continue
- *Interface stability* - A sudden change in the interface of VDU server will not affect the application
- *Flexibility* - Application can be modified and properly tested over time to comply with any changes on the server
- *Transfer speed* - Transferring files is limited only by the speed of the local network

7.1.1 Specifications and analysis

In this section, the formal VDU server API specifications, analyzed in chapter 4, are an important point of reference. As this specification clearly states, the VDU server communicates over the HTTP protocol in TLS[2]/HTTP channel and provides a REST API interface to interact with. Knowing this, the VDU server can be simplified down to a program, that:

[1] Graphical User Interface
[2] Transport Layer Security

- *Hosts an HTTP server with a secure TLS channel*
- *Is able to read and modify files*

Alternatively, in order for the simulated server to behave just like a real VDU server, it needs to simulate every endpoint of the real VDU server. The exact implementation of either server is not relevant for this purpose.

7.1.2 Design

Given the simplistic nature of the program requirements, in order to save time while sacrificing little to no functionality, I have decided to create the simulated VDU server using Python[3].

Python

Python is an interpreted, simple, easy to learn programming language with emphasis on readability of its syntax. It is considered a high-level language, allows for object-oriented programming and dynamic typing. A program written in the Python programming language is referred to as a Python script, since it is interpreted using the Python interpreter, which comes equipped with feature-rich default libraries that are completely able to cover the requirements for a simulated VDU server, as described by [17]. These facts make Python a great programming language choice for the implementation of the server.

Simulating endpoints

While knowing the exact implementation of the server would make simulation more precise, for the purposes of this thesis it was unknown. Thanks to the properly detailed specification and description of each endpoint, simulating every single endpoint was still possible.

7.1.3 HTTPS/TLS support

Knowing that the real server uses HTTPS protocol to communicate with the client, the simulated server had to support this as well. In order to start a

According to [19],

7.1.4 Implementation

7.1.5

7.2 Modifying the application

7.3 Unit testing

[3] https://www.python.org/

Chapter 8

Conclusion

[[Evaluation of progress etc.]]

The result application was released, and I published the source code as open-source on GitHub[1].

[1] https://github.com/

Bibliography

[1] Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN 0-13-235088-2.

[2] *Desktop Operating System Market Share Worldwide* [online]. Statcounter GlobalStats, 2021 [cit. 2020-03-09]. Available at: https://gs.statcounter.com/os-market-share.

[3] *Dokan - User mode file system library for windows with FUSE Wrapper* [online]. ISLOG, 2021 [cit. 2020-03-31]. Available at: https://dokan-dev.github.io/.

[4] Sharkey, K., Coulter, D., Jacobs, M. et al. *File Handles - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-10]. Available at: https://docs.microsoft.com/en-us/windows/win32/fileio/file-handles.

[5] Jacobs, M., Sharkey, K., Coulter, D. et al. *Files and Clusters - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-10]. Available at: https://docs.microsoft.com/en-us/windows/win32/fileio/files-and-clusters.

[6] *FUSE — The Linux Kernel documentation* [online]. The kernel development community, 2021 [cit. 2020-03-09]. Available at: https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[7] Stark, L. *Dokan-dev/dokany: User mode file system library for windows with FUSE Wrapper* [online]. ISLOG, 2021 [cit. 2020-03-20]. Available at: https://github.com/dokan-dev/dokany.

[8] *Microsoft/VFSForGit: Virtual File System for Git: Enable Git at Enterprise Scale* [online]. ©Microsoft, 2021 [cit. 2020-03-20]. Available at: https://github.com/Microsoft/VFSForGit.

[9] Zissimopoulos, B. *Billziss-gh/winfsp: Windows File System Proxy - FUSE for Windows* [online]. 2021 [cit. 2020-03-20]. Available at: https://github.com/billziss-gh/winfsp.

[10] Schofield, M., Sharkey, K. and Satran, M. *Handles and Objects - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-10]. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/handles-and-objects.

[11] *Cloud storage Icons - Free Download, PNG and SVG* [online]. Icons8, 2021 [cit. 2020-04-05]. Available at: https://icons8.com/icons/set/cloud-storage.

[12] *Local File Systems (Windows) | Microsoft Docs* [online]. ©Microsoft, 2016 [cit. 2020-03-10]. Available at: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/aa364407(v=vs.85).

[13] Whitney, T., Sharkey, K., Schonning, N. et al. *MFC Desktop Applications* [online]. 2021 [cit. 2020-03-09]. Available at: https://docs.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications.

[14] *An overview of HTTP - HTTP | MDN* [online]. Mozilla and individual contributors, 2021 [cit. 2020-03-26]. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[15] Robertson, C., Schonning, N., Tahan, M. A. et al. *C/C++ projects and build systems in Visual Studio* [online]. 2019 [cit. 2020-03-09]. Available at: https://docs.microsoft.com/en-us/cpp/build/projects-and-build-systems-cpp.

[16] Schofield, M., Walker, J., Coulter, D. et al. *Operating System Version* [online]. 2020 [cit. 2020-03-09]. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/operating-system-version.

[17] *What is Python? Executive Summary | Python.org* [online]. Python Software Foundation, 2021 [cit. 2020-04-10]. Available at: https://www.python.org/doc/essays/blurb/.

[18] Fielding, R. T. *Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)* [online]. 2000 [cit. 2020-03-26]. Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[19] Morrissette, M. *TUTORIAL: How to Generate Secure Self-Signed Server and Client Certificates with OpenSSL - The Devolutions Blog* [online]. 2020 [cit. 2020-04-10]. Available at: https://blog.devolutions.net/2020/07/tutorial-how-to-generate-secure-self-signed-server-and-client-certificates-with-openssl.

[20] *About Swagger Specification | Documentation | Swagger* [online]. SmartBear Software, 2021 [cit. 2020-03-26]. Available at: https://swagger.io/docs/specification/about/.

[21] *VFS for Git: Git at Enterprise Scale* [online]. ©Microsoft, 2021 [cit. 2020-03-31]. Available at: https://vfsforgit.org/.

[22] *MICROSOFT SOFTWARE LICENSE TERMS* [online]. ©Microsoft, 2019 [cit. 2020-03-10]. Available at: https://visualstudio.microsoft.com/license-terms/mlt031819/.

[23] Lee, T. G., Hogenson, G., Parente, J. et al. *Overview of Visual Studio | Microsoft Docs* [online]. 2019 [cit. 2020-04-09]. Available at: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide.

[24] *Windows 10 SDK - Windows app development* [online]. ©Microsoft, 2021 [cit. 2020-03-09]. Available at: https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk/.

[25] Radich, Q., Coulter, D., Jacobs, M. and Satran, M. *Windows Coding Conventions - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-11]. Available at: https://docs.microsoft.com/en-us/windows/win32/learnwin32/windows-coding-conventions.

[26] Sharkey, K., Satran, M. et al. *Directory Management - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-20]. Available at: https://docs.microsoft.com/en-us/windows/win32/fileio/directory-management.

[27] Zissimopoulos, B. *WinFsp* [online]. 2020 [cit. 2020-02-14]. Available at: http://www.secfs.net/winfsp/.

[28] Zissimopoulos, B. *Native API vs FUSE · WinFsp* [online]. 2021 [cit. 2020-03-20]. Available at: http://www.secfs.net/winfsp/doc/Native-API-vs-FUSE/.

[29] *GetSystemMetrics function (winuser.h) - Win32 apps | Microsoft Docs* [online]. ©Microsoft, 2018 [cit. 2020-03-16]. Available at: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getsystemmetrics.

[30] Schofield, M., Sharkey, K. and Satran, M. *Handle Limitations - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-11]. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/handle-limitations.

[31] Hollasch, L. W., Coulter, D., Kim, A. et al. *File systems driver design guide - Windows drivers | Microsoft Docs* [online]. 2020 [cit. 2020-03-09]. Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/.

[32] *[MS-ERREF]: NTSTATUS Values | Microsoft Docs* [online]. ©Microsoft, 2020 [cit. 2020-03-13]. Available at: https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55.

[33] Hudek, T., Coulter, D. and Sherer, T. *Using NTSTATUS Values - Windows drivers | Microsoft Docs* [online]. 2017 [cit. 2020-03-13]. Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-ntstatus-values.

[34] George, A. D., Wagner, B., Warren, G. et al. *File path formats on Windows systems | Microsoft Docs* [online]. 2019 [cit. 2020-03-20]. Available at: https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats.

[35] Schofield, M., Sharkey, K. and Coulter, D. *Structure of the Registry - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-16]. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/structure-of-the-registry.

[36] Schofield, M., Sharkey, K., Coulter, D. et al. *Semaphore Objects - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-18]. Available at: https://docs.microsoft.com/en-us/windows/win32/sync/semaphore-objects.

[37] Schofield, M., Hillberg, M., Guzak, C. et al. *Slim Reader/Writer (SRW) Locks - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-18]. Available at: https://docs.microsoft.com/en-us/windows/win32/sync/slim-reader-writer--srw--locks.

[38] Schofield, M., Sharkey, K., Coulter, D. et al. *Synchronization Functions - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-18]. Available at: https://docs.microsoft.com/en-us/windows/win32/sync/synchronization-functions.

[39] Bridge, K., Sharkey, K. and Satran, M. *Unicode in the Windows API - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-11]. Available at: https://docs.microsoft.com/en-us/windows/win32/intl/unicode-in-the-windows-api.

[40] Whitney, T., Sharkey, K., Coulter, D. et al. *Unicode Programming Summary | Microsoft Docs* [online]. 2016 [cit. 2020-03-12]. Available at: https://docs.microsoft.com/en-us/cpp/text/unicode-programming-summary.

[41] Sharkey, K., Satran, M. et al. *Volume Management - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-20]. Available at: https://docs.microsoft.com/en-us/windows/win32/fileio/volume-management.

[42] Radich, Q., Coulter, D., Jacobs, M. and Satran, M. *What Is a Window - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-03-15]. Available at: https://docs.microsoft.com/en-us/windows/win32/learnwin32/what-is-a-window-.