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

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2020/2021

Bachelor's Thesis Specification

Student: **Feranec Adam**
Programme: Information Technology
Title: **Application for Controlled Access to Remote Documents for Microsoft Windows**
Category: Operating Systems

Assignment:

1. Familiarize yourself with the requirements for secure access to documents in the Validated Data Storage (VDU) project. Explore the possibilities of virtual file systems and integration of applications into the desktop environment in the Microsoft Windows operating system.
2. Design a client application that will connect to the VDU repository and allow to access its content locally with a given one-time access token provided by the VDU system. The content will be accessed as a file in a virtual file-system and the client application will implement the access and version control. Also design automated tests of the application.
3. After consulting with the supervisor, implement the proposed application, including the automated tests.
4. Evaluate and discuss the results and publish the resulting software as open-source.

Recommended literature:

- An internal documentation of the Validated Data Storage project.
- VIRIUS, Miroslav. *Programování v C++: od základů k profesionálnímu použití*. Praha: Grada Publishing, 2018. Myslíme v. ISBN 978-80-271-0502-1.
- WinFsp: Windows File System Proxy. *GitHub* [online]. 2020 [seen 2020-10-26]. Available at [https://github.com/billziss-gh/winfsp]
- Dokany. *GitHub* [online]. 2020 [seen 2020-10-26]. Available at [https://github.com/dokan-dev/dokany]
- VFS for Git. *GitHub* [online]. 2020 [seen 2020-10-26]. Available at [https://github.com/Microsoft/VFSForGit]

Requirements for the first semester:

- Items 1, 2 and work started on item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rychlý Marek, RNDr., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: April 1, 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 19, 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

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[28] 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

Theory

2.1 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] Being in development for many years, to this very day, Windows provides many ways to create user applications, allowing developers to select from multiple programming languages at hand as well. The final choice of the C++ programming language, combined with the usage of C, came from the theis's aim. The aim of this thesis is to create an application, which integrates with the Windows enviroment, and even creates a virtual file system. For such low-level operations, which require close interacting with the operating system, C/C++ are the best languages, its just a matter of preference.

This chapter introduces application development for Windows using the Windows API, the Microsoft Foundation Class Library, and provides an overview of these technologies. The information is relevant for implementing the application in chapter 6.

2.1.1 Development Environment

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

Microsoft Windows Software Development Kit

The Microsoft Windows Software Development Kit (Windows SDK) is a required piece of software, used for developing and building applications for Windows. The Windows SDK contains all the libraries, headers, and tools required to design, implement, run, debug, and release Windows applications. Installing the Windows SDK allows the host computer to use debug versions of libraries, which do not translate to release versions of libraries.

Operating System Version

The operating system version, as described by [16], can be referred to as the build number of Windows, and is the application's target version. 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. The 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. An SDK can be compatibile

with an older version of the operating system. A good example of this is the Windows 10 SDK, which supports *Windows 7 Service Pack 1* as specified by [24]. Table 2.1 lists operating system versions for popular Windows desktop operating systems.

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

Microsoft Visual Studio

The Visual Studio integrated development environment (IDE), as described by [23], is a program developed by Microsoft and is ideal for Windows desktop application development. It includes a code editor with well written IntelliSense, debugging tools, theme customizations, support for third-party add-ons, and even a editor. Visual Studio is available in three different editions: Community, Professional, Enterprise. For students, Visual Studio 2019 Community (VS19) is the best option because, according to [22], it is free to use under Individual Licence, allowing any individual or student to work and develop their own applications. In Visual Studio, projects which work together are grouped under a *Solution*. Each project in a solution can be built for different operating systems, with different build tools, and with different project properties.

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.[15] It is usually best practice to develop under the latest version of build tools, which, at the time of writing are the *Build Tools v142*.

2.1.2 Windows API

The *Windows API*, also known as the *Win32 API*, is a massive, complex collection of headers and libraries programmed in the C programming language, containing many different functions, function prototypes, macros and documentation. The Windows API can be confusing to understand at first due to its uniqueness. To avoid this problem, this section aims to give a brief overview of what is important to know about Windows API, before implementing an application, from the bottom-up.

Integer Types

By standard, as specified in [25], a *Windows Word* is a 16-bit unsigned short integer, its data type is `WORD` and for historical reasons will always be guaranteed to be 16 bits long. A Double-Word is twice as long, 32-bit unsigned integer, `DWORD`. To support the new 64-bit architecture, a Quad-Word, `QWORD` is available. Additionally, Windows re-defines standard integers as their capitalized versions, such as `INT`, size of wich is architecture-specific, i.e., 32

bits on a 32-bit system. For precise sizes of integers, it is good practice to use a bit-speficic version, such as `INT32`. For unsigned integers, a *U* prefix is used. The use of capitalized standard data types is not that prominent, unlike *WORD*s, which are used very frequently in the Windows API.

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, as seen in the last example of Listing 2.1, is still a valid way to signify a pointer type while programming Windows applications, as mentioned in [25].

```
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

Code conventions

Windows uses *Hungarian Notation*[1], which adds semantical information to variable names in the form of prefixes.[25] 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, as shown in Listing 2.2.

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

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

Listing 2.3: The prototype of the GetSystemMetrics function. Source: citeWinGetSM

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

Character set

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

- ANSI[2] version, signified with the suffix *A*, i.e., `InternetOpenA`
- Unicode[3] 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 current character set.

Some newer functions do not support ANSI and only have the Unicode version available as stated by [42].

Strings

The usage of a s ties closely to the current project's character set, which is either defined by a macro or set up in the project settings in Visual Studio. 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, according to [43]. 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. 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. Listing 2.4 shows an example of all three types of defintions of a string.

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

Listing 2.4: An example of defining static strings

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*,

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

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*.[45]

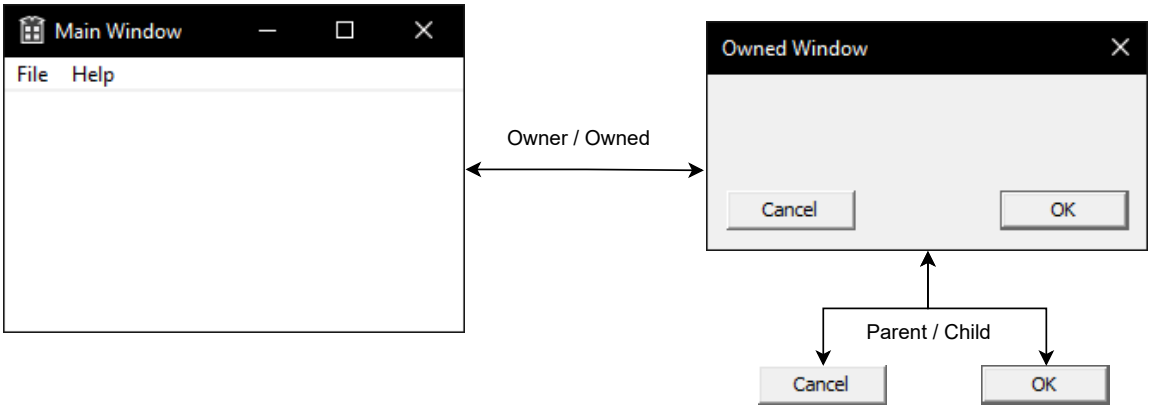

Figure 2.1: Example of owner/owned and parent/child relationships between windows.

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.[31]

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

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

type, and values `0x40000000 - 0x7FFFFFFF` define an information status type. This is easily checked with the `NT_SUCCESS(x)` macro, where `x` is NTSTATUS.[35][36]

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 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 (2.1.2) 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.

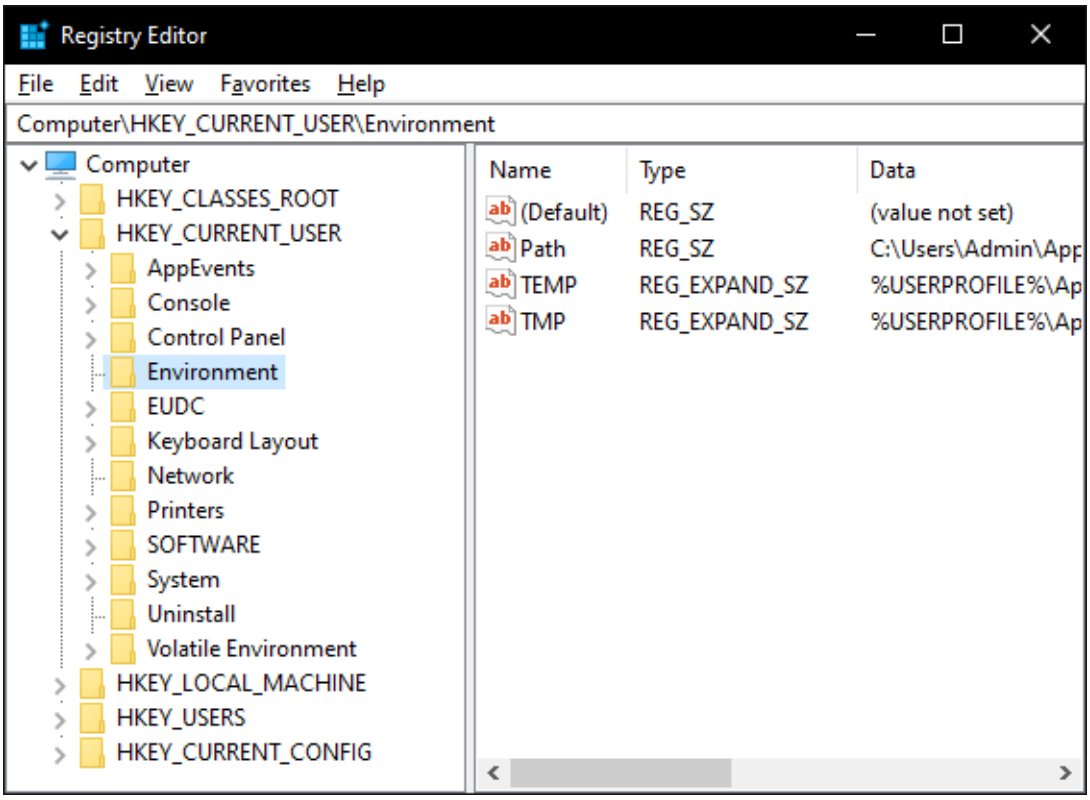

Figure 2.2: Browsing Windows registry using the Registry Editor.

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.[38] 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]]**.

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), listed by [41]. As this project makes use of SRW locks, this subsection will explain only those in the following.

As SRW lock is a simplified version of a semaphore, which is according to [39] described as 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. As specified by [40], 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 acqurie an SRW lock, it is not predictable which thread will acquire the lock in different modes. The lock is the size of a pointer, which means faster access but rather limited amount of information stored about the state of the lock. While being simple, it is sufficient enoguh to solve many thread synchronization problems, such as *„The Readers-Writers Problem*[5].

2.1.3 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 an object-oriented C++ library, which abstracts and wraps 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, as described by [13].

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

[5] https://www.u-aizu.ac.jp/~yliu/teaching/os/lec07.html

Listing 2.5 showcases an example of showing the main window (2.1.2) 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 of an MFC object, there are supportive functions like `GetSafeHwnd`[7], which return the internal object handle.

Coding conventions

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

Wrappers

Strings

Exception handling

[[Overview key MFC parts for this project]]

2.2 Virtual file systems

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[33]. 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 2.2.2.

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.

2.2.1 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.

File

Generally, in Windows, a *file* is a unit of data in a file system. A file is stored on a storage device[8] 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

[6] https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-showwindow
[7] https://docs.microsoft.com/en-us/cpp/mfc/reference/cwnd-class?view=msvc-160#getsafehwnd
[8] i.e. Hard Drive

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[9].[5]

Files in Windows are never accessed directly. Instead, applications on Windows can access a file through its handle (Section 2.1.2). 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]

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.[44] 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 (2.2.1) is the related data, and it can be organized into a directory or reside directly in the root of a volume.

File path formats

Windows uses the standard, traditional DOS[10] path format, which consists of the following components:

- *A volume or drive letter* - It is expected to be followed by the volume separator character[11]

[9] New Technology File System
[10] Disk Operating System
[11] The colon character (:)

- *A directory name* - The directory separator character[12] 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.[37]

Table 2.2: 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

2.2.2 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 creating of 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[13] subsystem 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.

Virtual file system software

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

[12] The backslash character (\)
[13] Input\Output

- 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.

- *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.[30][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.

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]

VFSForGit

Virtual File System for Git is software developed by Microsoft to enable Git[14] 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[15]
- *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]

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

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

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]

Conclusion

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 2.3 and 2.4.

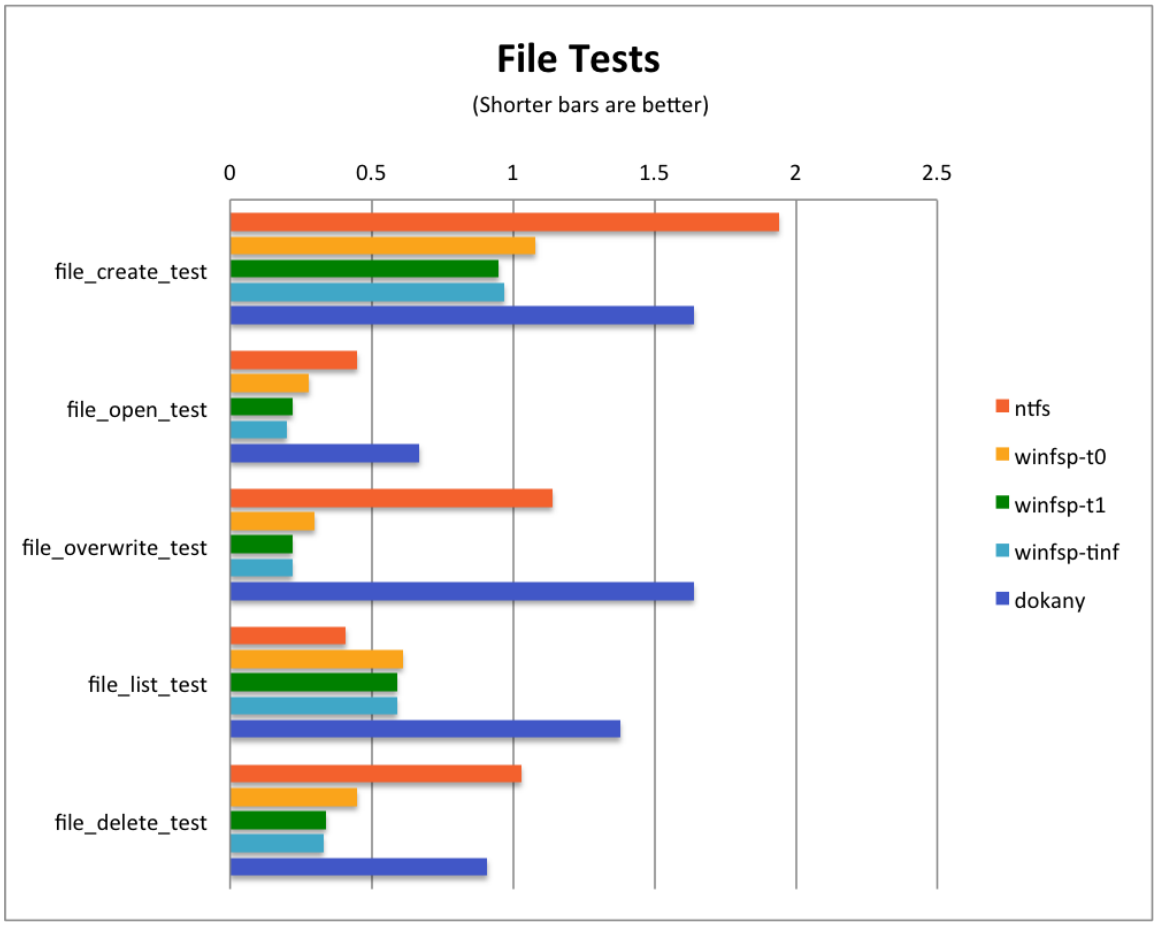

Figure 2.3: File comparison tests of WinFsp and Dokany. Source:[9]

Figure 2.4: Read and write comparison tests of WinFsp and Dokany. Source:[9]

2.3 Additional technologies

This section provides an overview of additional technologies used in this thesis, which do not fall under any specific category. It includes the Hypertext Transfer Protocol (HTTP), the Representational State Transfer (REST) and how it makes use of HTTP, along with a formal description format OpenAPI, used during the analysis in Chapter 4.

2.3.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol, as described by [14], is a simple, stateless communication protocol used for fetching resources. A resource can be anything that can be named, ranging from images, documents to generic files. HTTP is designed as a client-server type of protocol, in which a client and a server exchange information using HTTP messages. An HTTP message can be one of two types:

- *HTTP Request* - From client to server
- *HTTP Response* - From server to client

HTTP request

An HTTP request consists of the following elements, in order:

1. *Method* - Defines the requested operation with the resource
2. *Path* - Location of the resource
3. *Protocol version* - Version of HTTP
4. *Request headers* - Additional information about the resource
5. *Content* - Contains the content of the resource sent to the server

HTTP response

An HTTP response consists of the following elements, in order:

1. *Protocol version* - Version of HTTP
2. *Status code* - Defines the requested operation with the resource
3. *Status message* - Location of the resource
4. *Protocol version* - Version of HTTP
5. *Response headers* - Additional information about the resource
6. *Content* - Contains the content of the resource sent to the client

2.3.2 Representational State Transfer

According to [18], The Representational State Transfer represents an architectural style of developing RESTful web services, which allows the developer to take advantage of an already existing protocol. In the case of web services, this protocol is HTTP. REST 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

REST using HTTP

The key abstraction of information in REST is a *resource*. This definition is similar to the HTTP 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. The REST *endpoint* refers to the *path* of an HTTP request. The content of a resource is usually transferred as the *content* of an HTTP message. HTTP headers are useful for storing additional information about a resource, such as the file name, size of the content, encoding, etc. The HTTP method specifies the operation with a resource, which should conform to the REST API documentation of the server. An example of a simple REST API description is listed in Table 2.3.

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

Table 2.3: An example of a table of REST API endpoints

2.3.3 OpenAPI

The OpenAPI Specification is a description format for REST APIs. This format is handy for creating more formal descriptions of the entire API can be described with just a single file written in the OpenAPI format, which supports file formats of either YAML[16] or JSON[17]. According to [20], the OpenAPI format is capable of describing:

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

As Listing 2.6 shows, the OpenAPI format is easily readable and understandable for both machines and humans. Additionally, many third or first-party services provide a way to visualize the API in a graphical, user-friendly format, i.e., the Swagger Editor[18]. An example of a rendered graphical representation of the VDU server's REST API is displayed in Figure 4.1.

```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 2.6: An example of an OpenAPI file in the YAML format

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

Chapter 3

Specification

The goal of this thesis was to design, implement and test a client side application for Windows operating system, which integrates with the Windows desktop environment. This application should be able to connect to the VDU server and secure access to remote files present on the server, as they are being accessed by the user. The specification was provided by multiple sources:

- *Thesis specification* - Provided the exact steps this thesis should be taking
- *VDU API documentation* - Provided a detailed, non-formal description of the VDU server's API
- *Consultations with supervisor* - Provided more details, helped to narrow down design choices and testing ideas

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 of the VDU server. 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 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.

4.1.1 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[1] 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.

4.2 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.

[1] Hyper-Text Markup Language

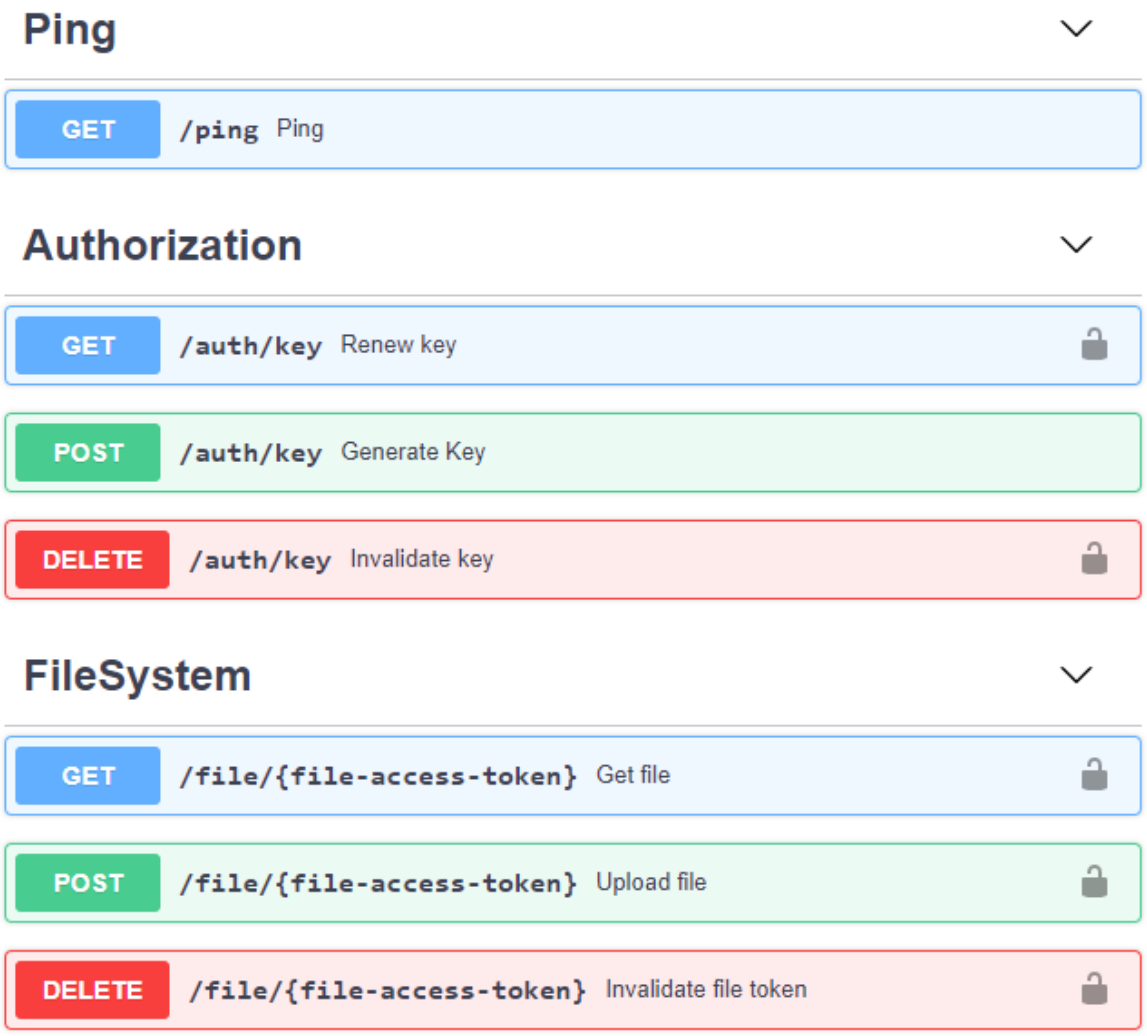

Figure 4.1: VDU REST OpenAPI 3.0 summary, rendered using the Swagger Editor, authentication requirement is signified by the lock icon.

4.2.1 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

4.2.2 Access

Both client and server must use a secure TLS[2] 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 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.

4.2.3 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.

[2]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 led me to an important realization. Windows Explorer, a built-in 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, as displayed in Figure 5.1. Using Windows File Explorer to provide

the functionality of file access-related actions is an 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 led 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.

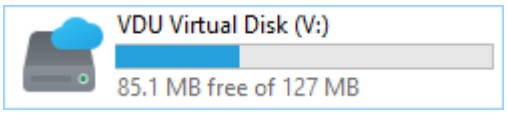

Figure 5.1: The VDU Virtual Disk as shown in the Windows Explorer.

5.1.2 Dialog design

After narrowing down all user actions, as shown in the use case diagram in Figure 5.2, the next step is to design the actual VDU client interface.

Considering the low amount of functionality required, I decided to design a simple *Extended Dialog Window* (dialog). The dialog, seen in Figure 5.3, 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.

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.

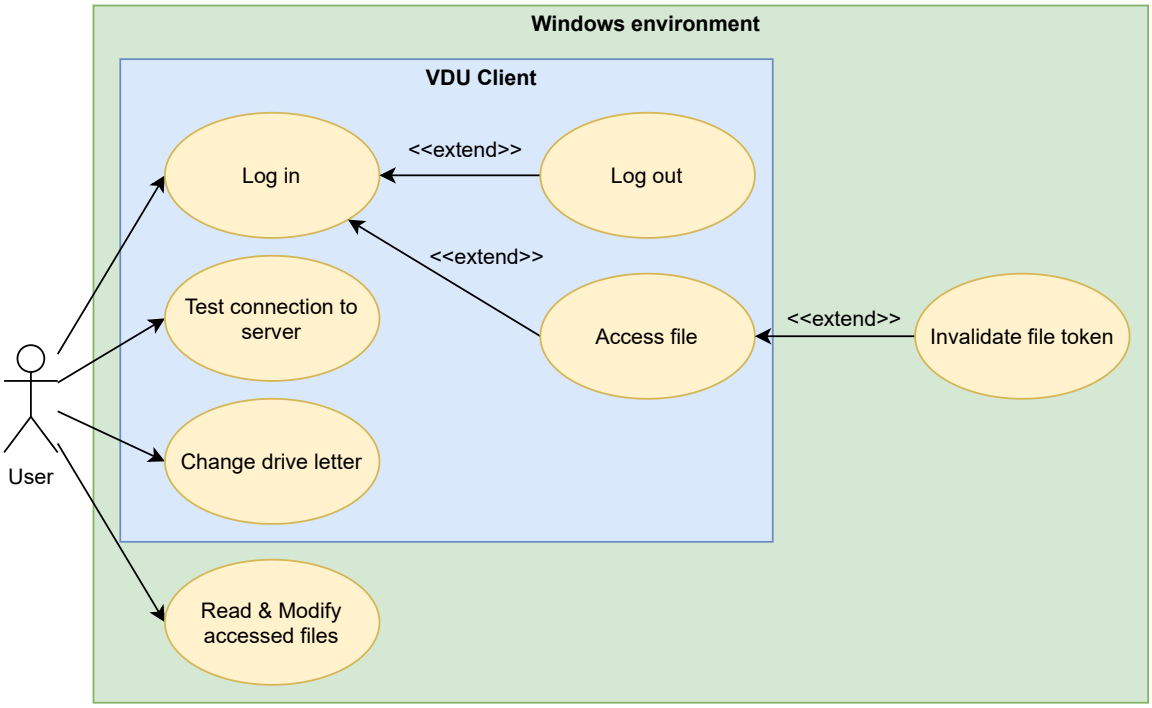

Figure 5.2: Use case diagram from the view point of an user of the VDU Client system and the Windows environment it resides in.

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 5.4, in the tray area.

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

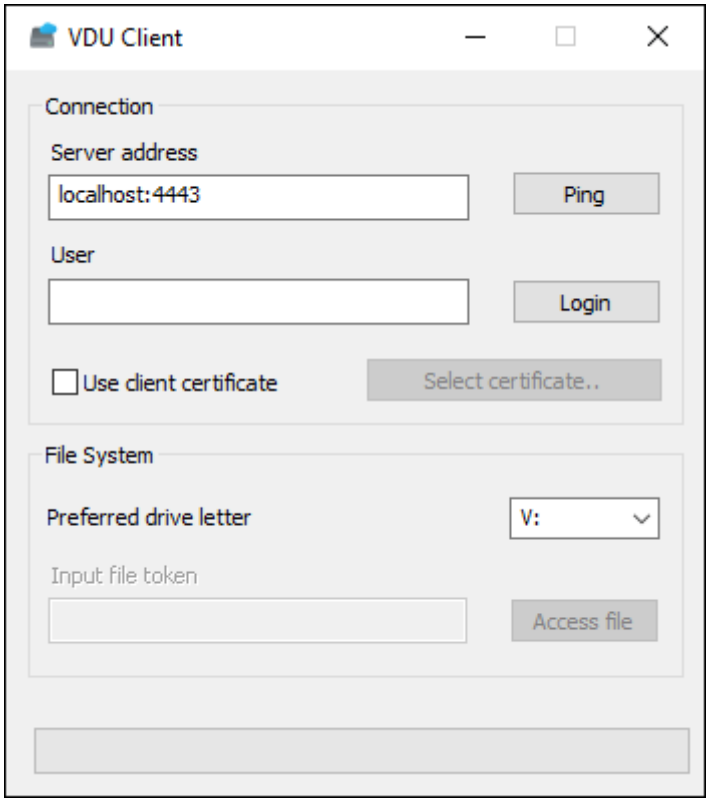

Figure 5.3: User interface of the VDU Client application.

Figure 5.4: Cloud Storage Icon, used as the main icon for VDU client application, Source:[11].

fix this problem, depicted in Figure 5.5, 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.

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 5.6.

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

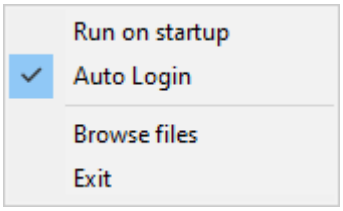

Figure 5.5: Tray menu of the VDU Client application.

Figure 5.6: Example of a tray tip, displaying the state of the application.

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 5.3, 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 5.7.

Windows notifications

For less important and rather informative actions, a Windows notification shows up as a response. Such a notification, displayed in Figure 5.8, appears, for example, after a successful download of a newly accessed file, along with an automatic startup of the assigned application to the file type. This lets the user know it was the VDU client which caused the application to open.

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

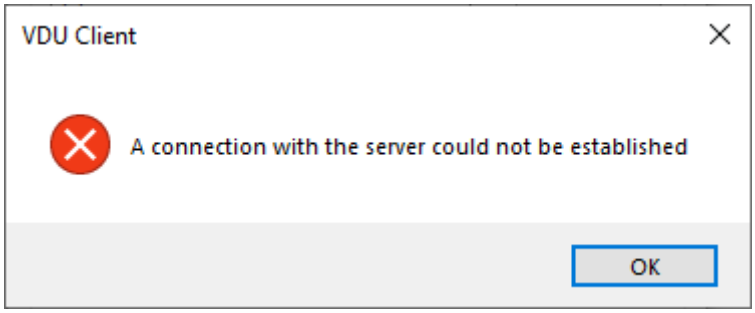

Figure 5.7: A message box window, displayed after the application fails to connect to a server.

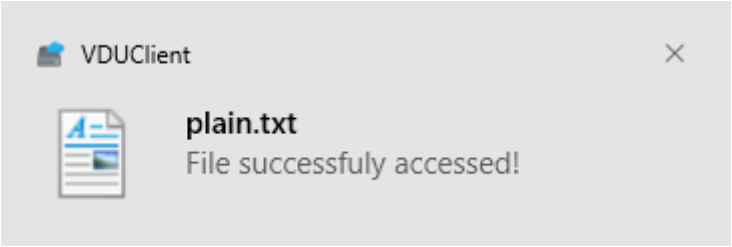

Figure 5.8: A windows notification, displayed after a successfully obtaining access to a file in the bottom right corner of the screen.

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 5.9, the concrete class structure and design's inspiration 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

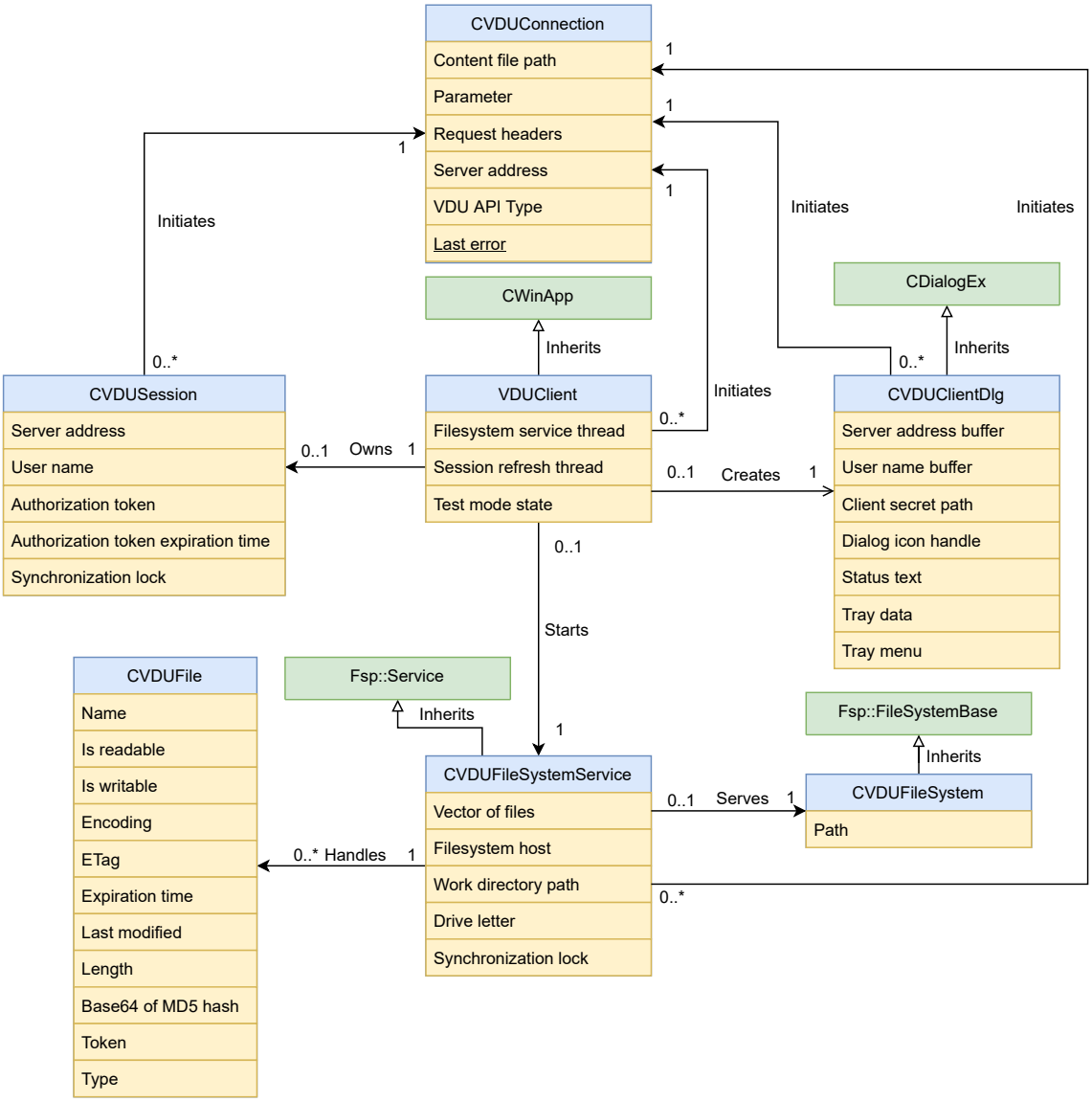

Figure 5.9: UML class diagram of the VDU Client application. [[**REDO IN THE PROGRAM**]]

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. All unsaved files will stay in this folder if the system crashes, 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. 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.

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 application's overall design. The entire implementation of the application was done using Visual Studio Community 2019, an IDE[1] which is further described in section 2.1.1. This chapter makes implicit use of the following macros:

- *WND* - Main window object (`CVDUClientDlg`)
- *APP* - The application object (`VDUClient`)

6.1 Back end

The back end implementation of the application corresponds to the class structure designed in Chapter 5. This section will cover the details of important parts of implementation, related to required application functionality.

6.1.1 Connection and session

The VDU connection is an extended wrapper of a regular HTTP conection to a VDU server, which handles all the overhead required to easily communicate with a VDU server using HTTP messages by sending a request and receiving a response. The VDU session is an abstraction of the state of communication between the VDU Client application and the VDU server, related to the current user. The session data most importantly consists of the user's *authorization token* and its *expiration date*.

Wrapping connections

Due to the state-less nature of HTTP, each request must be sent separarely from one another. Implementing requests regularly creates too much redundant code if the application needs to receive the response aswell.

My goal with creating a connection wrapper was to shift the generic and redundant HTTP connection related overhead into a single `CVDUConnection` class. Additionally, I recognized the only few variables which change from one connection to another, and made the class usable for communicating with every single VDU API endpoint. Thanks to this

[1] Integrated Development Environment

approach a request to the VDU server can be simplified into crating a connection object, as shown in Listing 6.1.

```
CVDUConnection conLogin(
    _T("127.0.0.1:4443"), //Server address
    VDUAPIType::POST_AUTH_KEY, //VDU API endpoint
    CVDUSession::CallbackLogin, //Callback function
    _T("From: John\r\n"), //Request headers
    _T(""), //Path parameter
    _T("C:\Client.crt")); //Path to content file
```

Listing 6.1: Example of instantiating a VDU connection wrapper class.

Threading connections

The application has by default only a single thread available, the thread which handles the user interface - the main thread. Processing connections on this thread would block the user interface from responding during the time of waiting for server's response.

I solved this problem by processing connections in separate, worker threads. Whenever the application needs to send a request to the server, it instantiates a new `CVDUConnection` object, and passes it as a parameter to a new thread - a connection thread. A connection thread starts it's execution at the beginning of a static func `CVDUConnection::ThreadProc`, which processes the connection and deletes the object from memory afterwards. Creating a thread using `AfxBeginThread`, as shown in Listing 6.2, is a non-blocking operation, which ensures that the main thread's execution flow will not be disrupted by issuing requests to the server.

```
LPVOID pCon = (LPVOID) new CVDUConnection(GetServerURL(),
             VDUAPIType::POST_AUTH_KEY, CVDUSession::CallbackLogin,
             headers, _T(""), certPath);

AfxBeginThread(CVDUConnection::ThreadProc, pCon);
```

Listing 6.2: Creating a new thread to process a connecton which sends a login request to the server.

[[FLOW CHARTS ARE GOOD]]

Thread synchronization

In a scenario where one or more worker threads are processing connections at the same time, all threads of the program are subject to a *data race*. The data race occurs due to shared access of the session data, and is capable of causing seemingly unreasonable errors, i.e. updating the authorization token right after a worker thread reads it from memory, resulting in the worker thread using an invalid authorization token for it's operation.

To solve this issue, I modified important parts of the code into *critical sections* using an SRW lock[2]. When a thread enters a critical section, no other worker thread can read or write into the session data without acquiring the lock in the *exclusive access* mode, as indicated in Listing 6.3. The main thread is excluded from this restriction, as it must not be blocked and the worst case scenario is only potentially outdated visual information.

[2]Slim Reader/Writer lock

```
CVDUSession* pSession = APP->GetSession();

//Blocking if already acquired, until released
AcquireSRWLockExclusive(&pSession->m_lock);
//Entered critical section

//Code which uses the session data exclusively
CString token = pSession->GetAuthToken();
...

//Leaving critical section
ReleaseSRWLockExclusive(&pSession->m_lock);
```

Listing 6.3: Example of a critical section implementation using an SRW lock.

Callback functions

In order to handle the results of the example login request demonstrated in Listing 6.2, the caller is allowed to specify a *callback* function. A callback function must be following the `VDU_CONNECTION_CALLBACK` prototype, declared in Listing 6.4. Every callback function has the following guarantees:

- *Executed asynchronously* - Executes in a worker thread
- *The parameter is the response* - The HTTP response can be `NULL` on failure
- *Exclusive access to sesion data* - To prevent data racing
- *Return value is thread exit code* - For synchronous operations

```
typedef INT (*VDU_CONNECTION_CALLBACK)(CHttpFile* httpResponse);
```

Listing 6.4: The prototype of a VDU callback function.

Refreshing authorization token

The VDU API states that an authorization token has an expiration time, after which the token is no longer valid. It is not mentioned what the expiration time span is. It could potentially be constant or it could be relative, it depends on the server.

To solve this issue by creating a permanent worker thread on application startup, which checks for expiration time every second, and sends a request to refresh the session once the expiration time span delta gets low enough. The reasoning behind the one second interval, instead of calculating the exact time a thread should sleep is, that there is no standard way to wake a thread up from sleep earlier, if the user does an unexpected action, such as, suddenly logging out.

6.1.2 File system

The VDU virtual file system is originally based on the *passthrough-cpp*[3] example file system made by *Bill Zissimopoulos*. The original example file system implemented accessing a given directory path via the virtual drive directly - a pass through file system. This was a perfect fit for this application, considering the VDU Client file storage design uses a folder in a very similiar sense. Basing the VDU virtual file system on it allowed me to spend more time perfecting the final system, as the example system covered a good chunk of the unrelated implementation overhead.

The virtual file system is implemented in the `CVDUFileSystem` class. The implementation consists of overriding virtual functions of the base `Fsp::FileSystemBase` class. The list of implemented virtual functions, along with a simple description according to [29], is the following:

- *GetVolumeInfo* - Volume information
- *GetSecurityByName* - File metadata and security descriptors
- *Create* - Creating a file
- *Open* - Opening a file
- *Overwrite* - Overwriting an existing file
- *Cleanup* - Situational file operations
- *Close* - Closing a file handle
- *Read* - Read bytes from file
- *Wite* - Write bytes to file
- *Flush* - Flush on disk
- *GetFileInfo* - Query file metadata
- *SetBasicInfo* - Set file attributes, file times
- *SetFileSize* - Change file size
- *CanDelete* - Whether or not can file be deleted
- *Rename* - Renaming a file
- *GetSecurity* - File's security descriptor
- *SetSecurity* - File's security descriptor
- *ReadDirectory* - Reading directory data
- *ReadDirectoryEntry* - Listing through directory contents

This file system is managed by the VDU file system service. The service is implemented in the `CVDUFileSystemService` class and holds all the information about VDU files, the file system status, the virtual file system drive and others. Most importantly, it implements functionality of transferring files between the client and the server and provides it to other parts of the application, including the file system itself.

[3] https://github.com/billziss-gh/winfsp/blob/master/tst/passthrough-cpp/

File integrity

Accessing a remote file via its file token triggers a download of the file from the VDU server to the local machine. The VDU client loads all the response headers and starts downloading the file. The file is at first downloaded into the current Windows user's temporary directory with a temporary name, prefixed the three letters *vdu*. After the download is finished, the application should verify the file's integrity to confirm it has been downloaded from the VDU server successfuly, without modifications.

This is achieved by creating an MD5 hash of the file's contents, and encoding the raw 16 bytes of hash data into the Base64 format. This format corresponds to the format used in the `Content-MD5` header of the server's response. If both hashes match, the file integrity has been proved, the file is registered and is moved into the applications work directory, available to be accessed by the user throguh the virtual file system.

Read-only files

VDU files can have a property, which diallows them to be uploaded to the server and thus, any modification - they are read only. While a file in Windows can have a read-only attribute set, many programs simply clear the attribute or ignore it completely. Modifying read-only files, whether by mistake or intention could lead to confusion and waste of bandwidth via requests, which will be denied by the server.

A decent approach to this issue is to disallow programs from acquiring a file handle, if the handle would have access to write to the file. The process of acquiring a handle to an existing file is handled in the `Open` function of the file system implementation.

Uploading files

Normally, local VDU files have to be manually uploaded to the VDU server every time a significiant enough change is made, to justify this effort. This makes it very difficult and annoying for the user to keep up with the changes and do a repetitive task over and over.

To automate this process, local VDU files, present in the virtual file system, will only be uploaded to the server if a change to the files contents is detected. If a change is detected, the file system service will upload the file in the background without the need of interaction of the user. If an error happens during the upload, the user is notified via a message box, explaining what went wrong. An upload request can potentially result in the file token being invalidated by the server, to which the application responds by removing the file locally as well, while notifying the user about this occurence.

Detecting file changes

A VDU file can be changed in many ways, via many different applications. Each application could use a slightly different method to modify the files. This makes it difficult to find a reliable way to detect the exact moment, when a file has changed, without repeatedly testing the file for changes on a timer - a very ineffective approach, which takes more processing power the more files are present in the virtual file system.

Detecting changes in a file is simple - create a new MD5 hash of the file and compare it to the one acquired from the VDU server. The problem is timing. The best time to detect a file change is instantly, and the best place for that is directly in the file systems implementation. A file can be changed in three basic ways:

- *Renaming* - Changing the file's name, and or extension
- *Replacing* - Drag and drop; overwriting the file with another file
- *Direct modification* - Opened and modified by some other application

Renaming is easy to detect, if a file is about to be renamed, the `Rename` virtual function of the virtual file system gets called. Inside this function I intercept this call and handle the detection.

Replacing happens, for example, with drag and drop operations. The exact process of this varies from application to application which handles this process, but generally, an application which replaces files creates a temporary file, writes new contents into that temporary file, and renames the temporary file to the original file name. Notice, that it makes use of the renaming functionality. To differentiate user-triggered renaming and renaming caused by application overhead, I use the `ReplaceIfExists` parameter of the `Rename` virtual function of the virtual file system. This parameter is `False` for overhead renaming, and `True` for user-triggerred renaming from Windows File Explorer. While the overead renaming could be used as a detection vector for file changes, I decided to use a more efficient approach, which solves the last way of file changing as well.

Direct modification detection stems from the cycle of modifying files by applications. This cycle, on its most basic level cosists of the following steps, which can be interctepted in the virtual file system:

1. *Open file* - Acquire file handle
2. *Modify the file* - Use the handle to modify the content
3. *Close file* - Close the handle

No application wants to keep a file handle for too long, as it would potentially prevent other applications from accessing this file. Thats why, instead of intercepting in the middle where the data is still being written, intercepting the end of the modification, when closing the file handle is the best spot. The virtual function `Close` provides the handle, which is a about to be closed via the `FileDesc0` parameter. To figure out, whether or not is this file one of the VDU files, the `GetFinalPathNameByHandle`[4] function provides the file name of the file this handle belongs to, which can be checked against the internal vector of VDU files. However, it is important to note, that this approach detects *every* handle that belongs to a VDU file. If an application intents to only read, and opens a read-only handle to a file and then closes it, it essentially creates a false-positive, as handles without explicit writing rights can not modify a file.

To solve this false-positive, the application needs to figure out whether or not does a handle have writing rights. The internal Windows API function `NtQueryObject`, according to [34], allows to query the `GrantedAccess` of a handle.

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

Invalidating file tokens

Drag-and-drop compatibility

Custom drive icon and label

For better visual clarity and easy recognition of the difference between a real drive and a virtual one, created by the application, I implemented a simple solution, which upon mounting a virtual drive to a drive letter, sets the icon to match the VDU Client main icon, as displayed in 5.4, and incude a descriptive drive label to match. To change a drives icon, all it takes is to create a registry key as a subkey to the *Explorer key*, which Windows File Explorer uses for its various settings. The created key's name has to be equal to the drive's letter. Inside this key the 'DefaultIcon' subkey's default value specifies the custom icon, and 'DefaultLabel' subkey's default value specifies the custom label. This is true for all Windows versions. The problem is, where is the Explorer key located. Accoring to [27], for Windows versions other than *Windows 2000*, which is this application's case, the key is located under the `HKEY_LOCAL_MACHINE` root key, and requires administrator permissions to be written into.

To avoid this, the key used in the Windows 2000 option's case can be used and is working as intended, with a modification. According to [32], the `HKEY_CLASSES_ROOT` key can be swapped for the `HKEY_CURRENT_USER\Software\Classes` key, if the intent is to write into the interactive user's settings, which is the exact case of this application. An example of implementation using this modified registry path to enable a custom icon is shown in Figure 6.5.

```
CRegKey key;
if (key.Create(HKEY_CURRENT_USER,
    _T("SOFTWARE\\Classes\\Applications\\Explorer.exe\\"
    "Drives\\Z\\DefaultIcon")) == ERROR_SUCCESS)
    {
      //Acquire the .exe path, containing the icon
      CString moduleFilePath;
      AfxGetModuleFileName(NULL, moduleFilePath);
      //Select the first icon, set the default value of key
      key.SetStringValue(NULL, moduleFilePath + _T(",0"));
      key.Close();
    }
```

Listing 6.5: Implementing a custom drive icon for drive Z, using without administrator permissions.

6.2 Front end

The front end of the application is the user interface, as it was designed in Chapter 5, and it is divided into the application's dialog window, and the Windows environment part. In the following subsections, the word *window* refers to all types of windows, e.g. button, combo-box, check-box, the application window, unless specified otherwise.

6.2.1 Dialog window

The dialog window was implemented using MFC[5] in Visual Studio, which allows creating dialong interfaces in a schematic-like format, as dispayed in Figure 6.1. Each designed

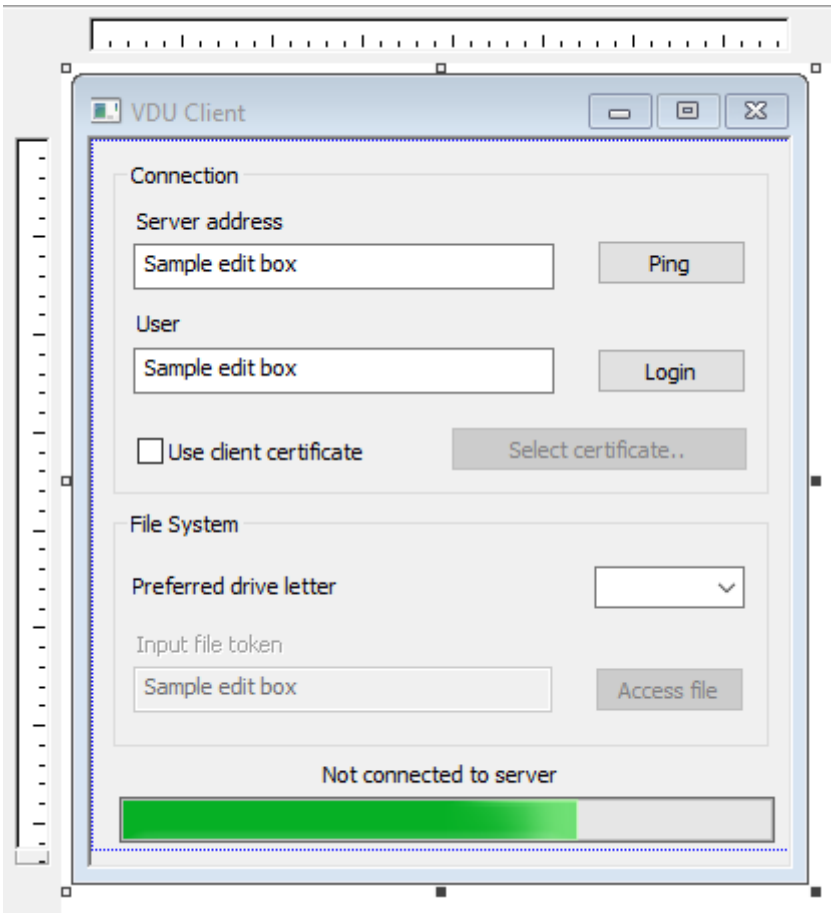

Figure 6.1: The VDU Client dialog window user interface, as designed in Visual Studio dialog editor.

window serves as a template for programatically created windows of the same type, and has a unique identifier - *name*. When creating a window, a template can be specified to guide the design of this window. As shown in Listing 6.6, the `Create` function of `CDialogEx` is uses the designed window template specified as the first parameter.

```
1 CVDUClientDlg* pDlg = new CVDUClientDlg();
2 pDlg->Create(IDD_VDUCLIENT_DIALOG, AfxGetMainWnd());
```

Listing 6.6: Creating the extended dialog window of VDU Client

Event handlers

In Windows, whenever a window is interacted with it receives a message, which can be handled and responded to. MFC provides a way to implement a function, which gets called

[5]Microsoft Foundation Class library

when a specific message is received - an *event handler*. For each child window of the dialog, including itself, I implemented one or more event handlers to assure functionality of the interface. An example of an event handler, along with it's mapping to a message and impelementation is displayed in Listing 6.7.

```
//Message mapping for dialog
BEGIN_MESSAGE_MAP(CVDUClientDlg, CDialogEx)
    ON_BN_CLICKED(IDC_BUTTON_PING, &CVDUClientDlg::OnBnClickedPing)
END_MESSAGE_MAP()

void CVDUClientDlg::OnBnClickedPing()
{
    TryPing();
    ...
}
```

Listing 6.7: Example of implemented and mapped click event handler

Tray

The core of interacting with the Windows tray area, and creating a tray icon, is the `Shell_NotifyIcon`[6] function of the Windows API. The information about tray icon is stored in `m_trayData` inside `CVDUClientDlg`. This data is modified and sent to the function whenever an operation with the tray icon is required. I divided interactions with the tray area into four parts:

- *Create* - `Shell_NotifyIcon(NIM_ADD, &m_trayData);`
- *Update tip* - `Shell_NotifyIcon(NIM_ADD, &m_trayData);`
- *Send notification* - `Shell_NotifyIcon(NIM_ADD, &m_trayData);`

To make it seem like the window gets hidden to tray, I override the system command `SC_CLOSE` to minimize and hide the window instead using the . 6.8

```
void CVDUClientDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == SC_CLOSE)
    {
        ShowWindow(SW_MINIMIZE);
        ShowWindow(SW_HIDE);
        return;
    }

    CDialogEx::OnSysCommand(nID, lParam);
}
```

Listing 6.8: Overriding system close command to hide the dialog window

[6] https://docs.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shell_notifyiconw

Message box

6.2.2 Windows environment

This subsection covers features, which tie closely to the Windows environment, such as the use of the tray area and the automatic start up of the application when the Windows user logs in.

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 VDU Client case, 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 better scalable.

7.1 VDU server simulation

Due to the network connectivity 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, the risk of flooding the production server due to 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 respond to the client's requests. This approach comes with several advantages:

- *Independence* - Even if the VDU server stops working, testing can continue
- *Interface stability* - A sudden change in the interface of the 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 of 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 its syntax's readability. 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 the simulation more precise, it was unknown for the purposes of this thesis. Thanks to the properly detailed specification and description of each endpoint, simulating every single endpoint was still possible.

7.1.3 Design

7.1.4 Implementation

7.1.5 Supporting HTTPS/TLS

[[Should I include how TLS works in the theory part? Certificates, CAs, asymmetric encryption, and stuff seems a bit too much maybe?]] Due to the fact that the real server uses HTTPS protocol to communicate with the client, the simulated server should be supporting the secure channel connection as well. In order to start the Python HTTP server in a way, which allows for HTTPS connections with the possibility of accepting client certificates, it required three important parts:

- *Certificate authority* - used to generate certificates
- *Server certificate* - contains the server's public key
- *Server key file* - contains the server's private key

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

It would be wasteful to acquire certificates from legitimate authorities, as the simulated VDU server will not be used for any purposes other than testing. Instead, I decided to solve this problem by creating a testing certificate authority (CA) using OpenSSL[4]. Upon creating my own testing CA, I used OpenSSL again to issue a testing certificate for the simulated VDU server. This operation generated both a public and a private key.

With CA and the server certificate generated, I wrapped the Python HTTP server's socket to use them and not require a client certificate from incoming connections. This operation allows the VDU application to successfully connect to the simulated server, assuming the application will ignore the invalidity of the CA. How I modified the application to support this is described in section 7.2.2. This subsection was created and implemented with the help of [19].

7.2 Modifying the application

By default, the application can only be controlled by the user using its user interface. This fact makes it difficult to test the application in an automated manner - issuing commands to the GUI[5] is not practical. A much better solution for this problem is to create a special mode in which the application is able to be launched - the *testing mode*.

7.2.1 Testing mode

7.2.2 Ignoring untrusted certificates

7.3 Automated testing script

7.3.1 C

[4] https://www.openssl.org/
[5] Graphical User Interface

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] Schofield, M., Sharkey, K. and Satran, M. *Assign a Custom Icon and Label to a Drive Letter - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-04-19]. Available at: https://docs.microsoft.com/en-us/windows/win32/shell/how-to-assign-a-custom-icon-and-label-to-a-drive-letter.

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

[29] Zissimopoulos, B. *WinFsp Tutorial · WinFsp* [online]. 2021 [cit. 2020-04-17]. Available at: http://www.secfs.net/winfsp/doc/WinFsp-Tutorial/.

[30] 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/.

[31] 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.

[32] Schofield, M., Sharkey, K. and Coulter, D. *HKEY_CLASSES_ROOT Key - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-04-19]. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/hkey-classes-root-key.

[33] 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/.

[34] *NtQueryObject function (winternl.h) - Win32 apps | Microsoft Docs* [online]. ©Microsoft, december 2018 [cit. 2020-04-19]. Available at: https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryobject.

[35] *[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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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-.

Appendix A

Contents of the included storage media

Appendix B

Manual