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, C++, WinFsp, MFC, file system, remote access, integration, Windows API, Python.

Kľúčové slová

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

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.

Rozšírený abstrakt

Táto práca sa zaoberá návrhom a implementáciou aplikácie, ktorá sprístupňuje vzialené dokumenty na serveri Vzdialeného Dátového Úložiska. [[**Continue RA**]]

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. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....
Adam Feranec
May 4, 2021

Acknowledgements

I would like to thank my supervisor RNDr. Marek Rychlý, Ph.D. for valuable feedback, numerous consultations, professional support and patience.

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 cloud drive service. All that is required is for the user to have an internet connection and log in to 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 cause a lot of overhead as the number of accessed files increases, and if the number of users increases as well, it could lead to a conflict of the content of the files.

Controlled access to remote documents and version control provides a solution to these issues. Ideally, it should be present on the server's side with a helping application running on the user's device. In this thesis's case, the server belongs to a project, which will be referred to as The Validated Data Storage project (VDU). This thesis analyzes the requirements of access to the VDU server and creates a client-side application for Microsoft Widnows[2] from the ground up. It implements a virtual file system, present on a virtual disk, integrated into the desktop environment of Windows, improving the user experience. This type of integration allows to seamlessly view and modify files stored in the cloud as if they were present on a local disk of the machine running the application. All of that, without the need to manually download or upload after each modification. This creates a user-friendly environment where all operations on remote files are intuitive to use and do not require an excessive tutorial or great understanding of the underlying technologies. For example, to access a file, all that is required is the file access token, and the file immediately opens and is ready for use. The custom virtual file system will do all the work in the background. The application is published as open-source on GitHub[3]. **[[Popis co vyvinout]]**

[1] https://www.microsoft.com/en-ww/microsoft-365/word
[2] https://www.microsoft.com/en-us/windows
[3] https://github.com/coolguy124/vduclient

Chapter 2

Theory

This chapter serves as an introduction to the required technologies for this thesis, according to the specifications, while overviewing them. It intends to cover the development of applications for Microsoft Windows, files and various virtual file systems, the possibilities of integration with the Windows environment, and additional used technologies.

2.1 Development for Microsoft Windows

According to [43], Microsoft Windows is the most used desktop operating system, consistently having more than a 70% market share among operating systems across many years. Being in development for many years and thanks to its great backward compatibility, Windows provides many ways to create user applications, allowing developers to select from multiple programming languages. The choice of the C++ programming language, combined with the usage of C, came from the thesis's aim. This thesis aims to create an application that integrates with the Windows environment and even creates a virtual file system. For such low-level operations, which require close interaction with the operating system, C/C++ is the most effective programming language since they operate on the same level.

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 5.

2.1.1 Development Environment

This subsection focuses on all 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 software for developing and building applications for Windows. The Windows SDK contains all 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 the libraries, which do not translate to release versions of libraries.

Microsoft Visual Studio

The Visual Studio Integrated Development Environment (IDE), as described by [12], 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 customization, support for third-party add-ons, and even a graphical window 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 [18], 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 as stated by [24]. 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*.

Processor modes

According to [44], in the Windows operating system, the processor can run code in two modes:

- *Kernel-mode* – Privileged mode
- *User-mode* – Unprivileged mode

The processor can switch between these modes depending on what code is being executed. Hardware drivers, file system drivers, and the operating system kernel all run in the kernel-mode. All components in the kernel-mode share the same address space, have privileged access to the entire system, and even access each other's data. In user-mode, every process started on Windows has its own virtual address space. The virtual address space is private for that process, ensuring no other process can access it - every application runs in isolation.

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. This section aims to give a brief overview of what is important to know about the Windows API before implementing an application from the bottom up to avoid this problem.

Integer Types

According to [23], 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`, the size of which 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-specific 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 frequently used 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*. As seen in the last example of Listing 2.1, using the standard star symbol is still a valid way to signify a pointer type while programming Windows applications, as mentioned by [23].

```
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.[23] 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, used in Windows API development

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*, as specified by [34], 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 object table of the operating system. An application

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

can obtain a handle through various Windows API functions, depending on the object the application is trying to access.

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.[33]

Character set

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

- ANSI[2] version – signified with the suffix *A*, i.e., `InternetOpenA`
- Unicode[3] version – signified with the suffix *W*, i.e., `InternetOpenW`
- Generic version – adaptive 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 [1].

Strings

The usage of strings ties closely to the current project's character set. 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 [45].

Both prototypes and functions provide the programmer with a method to work with strings and adapt to the preferred character set automatically. Those functions are 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 2.3 shows an example of all three types of definitions 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 2.3: An example of defining static strings

Windows

A window in terms of Windows API described by [22] is a programming construct which:

- *Occupies some portion of the screen*
- *Can change its visibility at any given moment*

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

- *Knows how to draw itself*
- *Responds to events from the user or the operating system*

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

Relative to other windows, a window can have relationships. If another window creates a new 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.*[22]

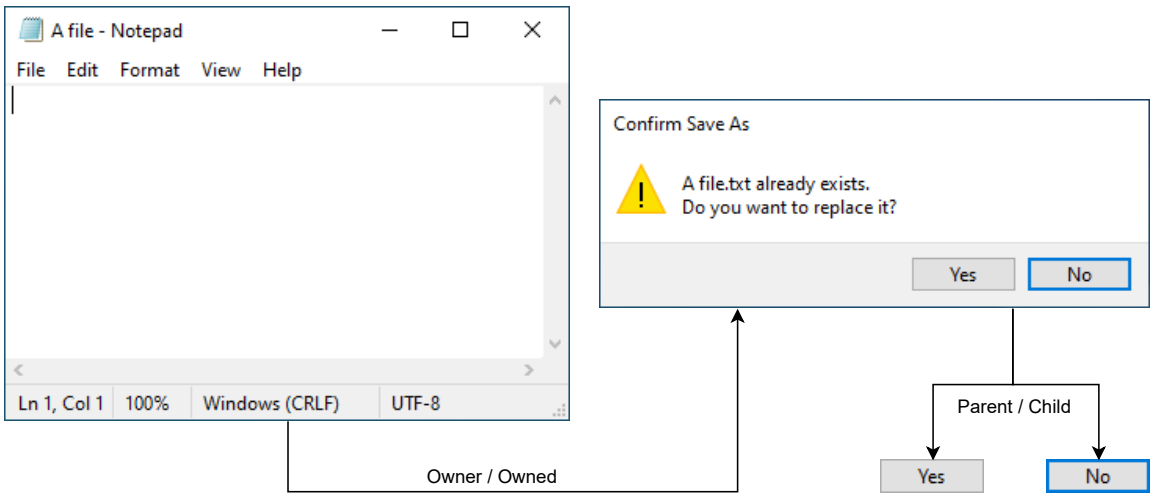

Figure 2.1: Example of owner/owned and parent/child relationships between windows.

Registry

The Windows registry is a hierarchical database containing data critical for the correct function of the operating system's services and applications that run on it. The registry data structure, as described by [27], 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. The value data types are standard Windows types, such as a double-word or a zero-terminated string. There are several predefined keys, and according to [28], Each serves a different purpose, either for the operating system or for the services and applications that run on it. The predefined keys are always open and are noted by the `HKEY_` prefix. Figure 2.2 displays the Windows registry using the Registry Editor.

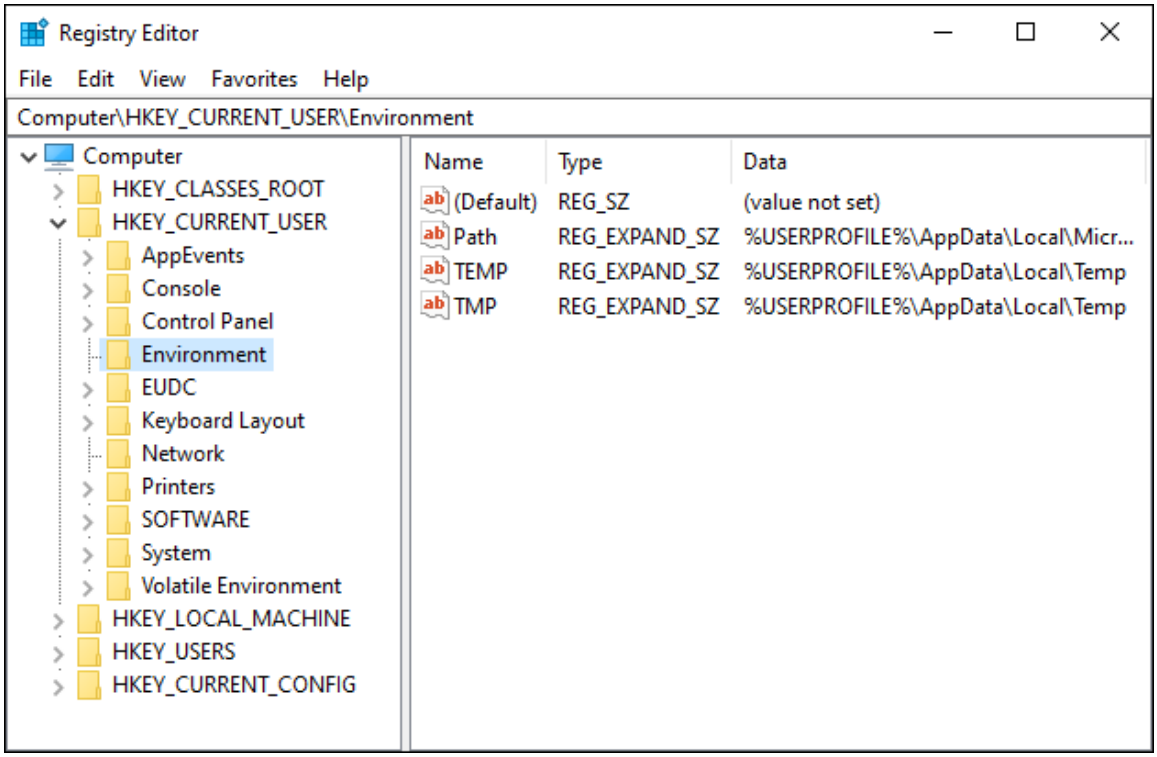

Figure 2.2: Browsing Windows registry using the Registry Editor.

The predefined keys which are important for Windows applications are the following:

- `HKEY_CLASSES_ROOT` – Definitions of document types and classes; shell information
- `HKEY_CURRENT_USER` – Preferences of the current Windows user
- `HKEY_LOCAL_MACHINE` – System configuration data
- `HKEY_USERS` – Default user and current user configuration
- `HKEY_CURRENT_CONFIG` – Differences between the current and standard configuration of the system

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 [30]. As this thesis makes use of SRW locks, this subsection will explain only those in the following.

An SRW lock is a simplified version of a semaphore, which is, according to [29], described as a synchronization object that 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 using the resource, another thread is allowed to use it. As specified by [25], 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 the 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 acquire 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 a rather limited amount of information stored about the state of the lock. While being simple, it is sufficient to solve many thread synchronization problems, such as *„The Readers-Writers Problem*[4].

Interprocess communication

There are multiple different methods of performing interprocess communication in Windows, such as named pipes, Windows sockets, mailslots, etc. This thesis specifically makes use of the *mailslot*.

A mailslot, according to [31], is a temporary pseudofile, only present in the system's memory. A mailslot can be created by a single process, which becomes the mailslot server, and only that process can read from the mailslot. Other processes can become mailslot clients by opening and writing into the mailslot file. This creates a dynamic where the process can check whether it is a server or a client by simply attempting to create a mailslot and possibly having a different behavior depending on the result. The mailslot can contain small messages with a maximum of 424 bytes, which is enough for simple many-to-one communication of Windows processes.

2.1.3 Microsoft Foundation Class Library

The Microsoft Foundation Class Library (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 [46].

Abstractions

As mentioned in previous sections, MFC allows for much easier desktop application development by abstracting and wrapping a large part of the Windows API, originally written in C, into the object-oriented C++ programming language. Listing 2.4 showcases an example of showing the main window using both APIs and an instance of abstracting the window handle away in favor of using a window C++ object. Calling the `ShowWindow`[5] 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`[6], which return the internal object handle.

[4] https://www.u-aizu.ac.jp/~yliu/teaching/os/lec07.html
[5] https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-showwindow
[6] https://docs.microsoft.com/en-us/cpp/mfc/reference/cwnd-class?view=msvc-160#getsafehwnd

```
//Windows API - Using C
HWND hMainWnd = CreateWindowW(...);
ShowWindow(hMainWnd, SW_SHOWNORMAL);

//MFC - Using C++
AfxGetMainWnd()->ShowWindow(SW_SHOWNORMAL);
```

Listing 2.4: Showing a window using Windows API and MFC

Coding conventions

All global, static MFC functions are marked with an `Afx`, prefix (Application Framework Extension). These functions provide support for managing the main window, creating new threads, acquiring the application instance, etc. Variable naming uses a similar scheme to Windows API.

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. The following sections contain an introduction to files, file systems, virtual file systems, and an overview of available third-party virtual file system software.

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, and 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[7] and consists of one or multiple streams of bytes, which hold related data, and a set of attributes that describe the file and its data. The file system manages it, and any application that wants to access, read, write, or execute a file or its attributes has to interact with its respectable file system to do so. A file must follow the file systems' rules, i.e., a file must have a unique name in its directory in NTFS[8].[10]

Files in Windows are never accessed directly. Instead, applications on Windows can access a file via a file handle. 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 is requested, if one process holds a handle to a file, a second process trying to open a handle to the same file might fail.[39]

[7] i.e. Hard Drive
[8] New Technology File System

File system

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 file systems supported by Windows, as described by [16]. have the following storage components:

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

A Volume is where the file system resides, is the highest level of organization in a file system, and has at least one partition, a physical disk's logical division.[41] For this project's purpose, only volumes with a single partition (simple volume) will be considered. Such a 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 that 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, which can itself be organized into a directory, and has no limitations on the number or capacity of files that it contains. Limitations of directories are defined by the file system itself and the capacity of the storage device.[40] 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 is the related data, and it can be organized into a directory or reside directly in the root of a volume.

Windows file systems

According to [8], a file system for Windows has to be implemented as a kernel-mode driver. Additionally, the driver has to be certified and signed by an authority. The standard supported file systems are, most importantly, NTFS[9], ExFAT, UDF, and FAT32, with NTFS being the default option used for the main local drive of the system.

On Linux, it is possible to create a file system for which, according to [13], the data is provided by a regular user-mode process or an application. This type of file system is referred to as a file system in userspace (FUSE), and it does not exist on Windows. Fortunately, there are still a few solutions to implementing such a file system, a virtual file system, in Windows.

2.2.2 Virtual file system

A virtual file system is an abstraction of a regular file system, and as noted by [37], any information and 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 or extend upon it. A virtual file system's power comes from integrating closely with the operating system - hooking into the system's internal file operating functions and handling them in its own way. That way, it can enforce any arbitrary rules on volume, directory, and file

[9] New Technology File System

management. To create a virtual file system without creating a kernel-mode driver, it is much preferable to use a third-party virtual file system software.

Virtual file system software

The virtual file system software allows a developer of a user-mode application to create a virtual file system. The software typically includes its own kernel-mode file system driver, which should be well tested and certified. This driver communicates with a user-mode library, which provides a virtual file system API. This is the API an application can use to create a virtual file system. According to [36], there two ways in which a modern virtual file system software can provide a virtual file system API:

- Native API
- FUSE Compatible API

A *Native API* ties closely to a single operating system, meaning it focuses on working with the intended system as seamlessly as possible.

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

A *FUSE Compatible API* allows for cross-platform compatibility with file systems in the user space created for Linux, with little to no changes to their implementation. Note that for Windows, the implementation of file systems is vastly different from Linux. Using a FUSE compatible API comes with compromises since some features are only present in Windows, i.e., volume labels, as described by [36].

- *Pros* – Cross-platform compatibility; easier development with a 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 a virtual file system software that includes a native API, as cross-platform compatibility is not a requirement. This would also mean using Windows-specific file system-related features and would be a step towards a better user experience.

Dokany

Dokany was created by *Hiroki Asakawa*, and as described by [6], it is one of the oldest yet still fully functional pieces of virtual file system software. It was created in 2007, and while undergoing a switch 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*: 32-bit, 64-bit

- *Supported desktop operating systems*: Windows 7 SP1 / 8 / 8.1 / 10
- *Open-source*: Yes
- *Provides a driver*: Yes

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 low-level C, as [3] specifies.

Virtual File System for Git

Virtual File System for Git (VFSforGit) was created by *Microsoft*, and as described by [7], it is a software developed by Microsoft to enable Git[10] at an 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[11]
- *Supported languages*: Git commands
- *Supported architectures*: 64-bit
- *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 does not 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, as specified by [19].

Windows File System Proxy

Windows File System Proxy (WinFsp) was created by *Bill Zissimopoulos*. As described by [5], it is a performant and stable piece of software, which allows to implement a virtual file system using either of its supported virtual file system API layers. The focus of WinFsp is on outperforming other software and compatibility with NTFS[12], the default file system of modern Windows. This allows for smooth integration with the Windows environment and virtual file systems, which use or extend NTFS.

- *Supported API types*: Native, multiple FUSE compatibility layers
- *Supported languages*: C, C++, DotNet

[10] https://git-scm.com/
[11] https://github.com/microsoft/VFSForGit/blob/master/Protocol.md
[12] New Technology File System

- *Supported architectures*: 32-bit, 64-bit
- *Supported desktop operating systems*: Windows 7 and above
- *Open-source*: Yes
- *Provides a driver*: Yes

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

Conclusion

The third-party virtual file system software of choice for the VDU Client application is WinFsp. The reasoning behind this choice is the following:

1. *VFSforGit* could not be used because of its harsh limitations or operating system compatibility and the focus on Git repositories.
2. *Dokany* was a great option. It is stable, well supported, and tested and has multiple compatibility layers. However, it does not outperform WinFsp, and it does not provide a C++ API library.

WinFsp's native API, which is even available for C++, allows for cleaner and easier to understand code and offers much better performance and optimization than Dokany. This is proved by various file system operation tests that compare versions of WinFsp with Dokany and NTFS. The performance comparison charts are displayed in Figure 2.3 and 2.4.

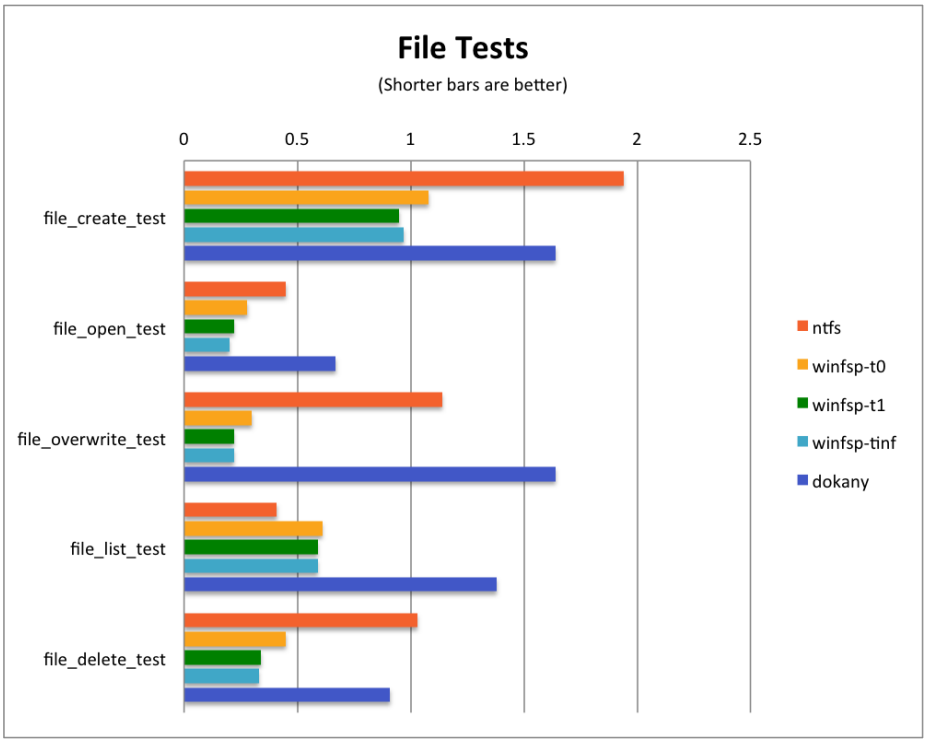

Figure 2.3: Performance comparsion of file tests of WinFsp, Dokany and NTFS. Source:[5]

Figure 2.4: Performance comparsion of read/write tests of WinFsp, Dokany and NTFS. Source:[5]

2.3 Additional technologies

This section provides an overview of additional technologies required by the specification and used for its analysis, design, and implementation.

2.3.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol, as described by [20], 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:

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:

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 [4], 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 a 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 2.1.

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 2.1: An example table of REST API endpoints with their respective descriptions.

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, which can be described with just a single file written in the OpenAPI format, which supports file formats of either YAML[13] or JSON[14]. According to [42], the OpenAPI format is capable of describing:

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

As Listing 2.5 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[15]. An example of a rendered graphical representation of the VDU server's REST API is displayed in Figure 3.1.

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

```yaml
#A simple documentation of a /ping endpoint
openapi: 3.0.0
info:
  version: '1.0'
  title: An amazing API
  description: 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 2.5: An example of an OpenAPI file in the YAML format

Chapter 3

Specification and analysis

This chapter will tackle the first step of creating an application – the analysis of the provided specification. Additionally, it shows the steps taken during an analysis and formalization of the provided documentation of the VDU server, including the formalization of the server's API, with an overview of the required technologies.

3.1 Requirements

This thesis aims to design, implement, and test a client-side application for Windows operating system, which integrates with the Windows desktop environment. This application should connect to the VDU server and secure access to remote files present on the server as the user is accessing them. Multiple sources provided the specification:

- *Thesis specification* – Provided the exact steps this thesis should be taking
- *Internal VDU API documentation* – A description of the VDU server's API
- *Consultations with supervisor* – Provided more details, helping to narrow down design choices and to test ideas

The internal VDU API documentation was provided as a document with a detailed description of every available endpoint of the API with additional information about how an application is supposed to connect to the server.

3.1.1 Endpoints

This subsection lists all available VDU API endpoints and provides an overview of their usages. Some endpoints require authentication of the user for successful access.

Ping

Used to test a connection to the server.

- *GET /ping* – Test connection to the server

Authentication

Used for authentication purposes.

- *POST /auth/key* – Authenticates a user by name or email. The client secret can be included in the content if necessary. Returns API key if it succeeds.
- *GET /auth/key* – Returns a new API key with a new expiration time, refreshes session
- *DELETE /auth/key* – Invalidates API key

File system

Used for remote file management. Where the `file-access-token` is a variable representing a file token obtained from the VDU web interface.

- *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 the server

Access

The client and server must use a secure TLS[1] channel to access the VDU API, while the server must have a valid server-side TLS certificate. This implies the usage of HTTPS[2] 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 a key obtained from the `POST /auth/key` endpoint. Each 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 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 one 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.

Version control

The VDU server handles the file version control system. The client application does not manage or control the version of a file. This version is noted as an `ETag` header, which can be any string. The server can change this header 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.

[1] Transport Layer Security
[2] Hypertext Transfer Protocol Secure

Communication

The communication between the client and the server uses the Hypertext Transfer Protocol, which goes over a secure TLS/HTTPS channel. The client initiates the communication with a request message to the target endpoint of the action that it is trying to perform. The server accepts this message, validates the authentication data for the required endpoints, and responds to the message with a response message to the client. An example of this communication is shown in Figure 3.1 and 3.2, where the client requests a file with the token `a` from the `/file/` endpoint using the `GET` method. The server accepts this request, implying that the authentication data in the client's request header is correct and responds with a response message containing all information about the requested file, including its name, length, type, content, etc.

```
GET https://127.0.0.1:4443/file/a HTTP/1.1
X-Api-Key: c57v7n34evuqjlyfmmo0miw3kkr5y4d8ndl58hy1sx6i7go2gpdi9ge17hlu1y1q
User-Agent: VDUClient 1.0, Windows
Host: 127.0.0.1:4443
Cache-Control: no-cache
```

Listing 3.1: An example of the contents of an HTTP request sent from the VDU client to the VDU server.

```
HTTP/1.0 200 OK
Allow: GET POST
Content-Location: plain.txt
Content-Length: 60
Content-MD5: 16EU4Wni3wdB4d7x370Ung==
Content-Type: text/plain
Date: Thu, 29 Apr 2021 19:29:00 GMT
Last-Modified: Thu, 29 Apr 2021 19:28:12 GMT
Expires: Thu, 29 Apr 2021 19:31:00 GMT
ETag: 4

This is the file contents, this is a simple plain text file.
```

Listing 3.2: An example of the contents of an HTTP response sent from the VDU server to the VDU client.

3.2 Formalization

Considering the provided documentation, formalization means creating an OpenAPI specification based on the documentation's plain text version. A formalized specification allows for better readability, understanding, development, and testing on the developer's side.

3.2.1 OpenAPI specification

Formalizing the provided documentation consists of reading and understanding all API endpoints, their access, usage restrictions, and manually creating an entry for each endpoint in a file of the OpenAPI format. Each entry has its own list of status codes, headers, and content, which the endpoint could potentially return. For the documentation of the VDU server's API, I used the Swagger Editor, which allowed me to document it as a REST API

more comfortably. An advantage of the online editor is that it can render the OpenAPI specification file in the HTML[3] 5 format, making the API easy to read and understand.

Conclusion

The result of the analysis was a file of the OpenAPI format `openapi.yaml`, which contains the formal descriptions of the VDU API endpoints. I discussed this information further with my supervisor, which allowed me to understand better how the API works and how it should interact. Figure 3.1 displays the summary of a rendered version of the formalized VDU server's API.

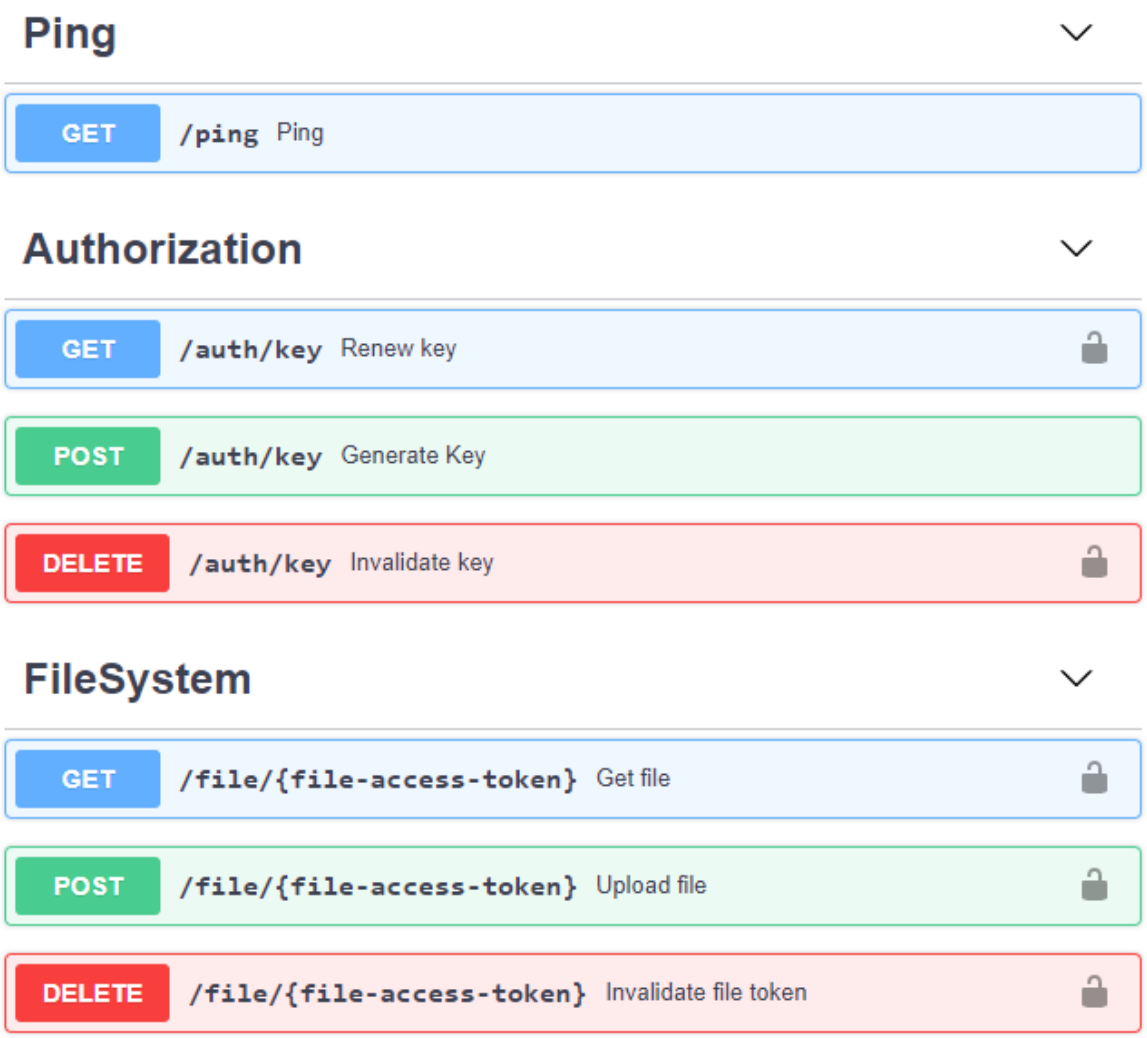

Figure 3.1: VDU REST OpenAPI 3.0 summary, rendered using the Swagger Editor, authentication requirement is signified by the lock icon.

[3]Hyper-Text Markup Language

Chapter 4

Design

This chapter aims to design the application based on knowledge from the specification and analysis of the requirements. The design of the application consists of designing the internal components and the user interface.

4.1 Internal components

Choosing the right design for the application's internal components is the key to creating a performant, reliable, and scalable program. Knowing that the application is to be implemented in C++, I designed the class structure before implementation.

4.1.1 Class structure

The concrete class structure and design's inspiration was the *Single Responsibility Principle* (SRP). As the creator of the principle, *Robert C. Martin* states in [14], SRP is a design principle for designing object-oriented classes or modules. The key inspiration of SRP for the class design was that a single class should only have a *single reason to change*. This approach allows each class to be developed independently from others and has led me to design the following classes for the application:

- *VDUClient* – Main class of application
- *CVDUClientDlg* – Instantiates and handles the dialog window
- *CVDUConnection* – Serves as a wrapper for communication 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 4.1 shows the class diagram of the application using the Unified Modeling Language (UML). This class diagram was created using Visual Paradigm[1].

[1] https://www.visual-paradigm.com/

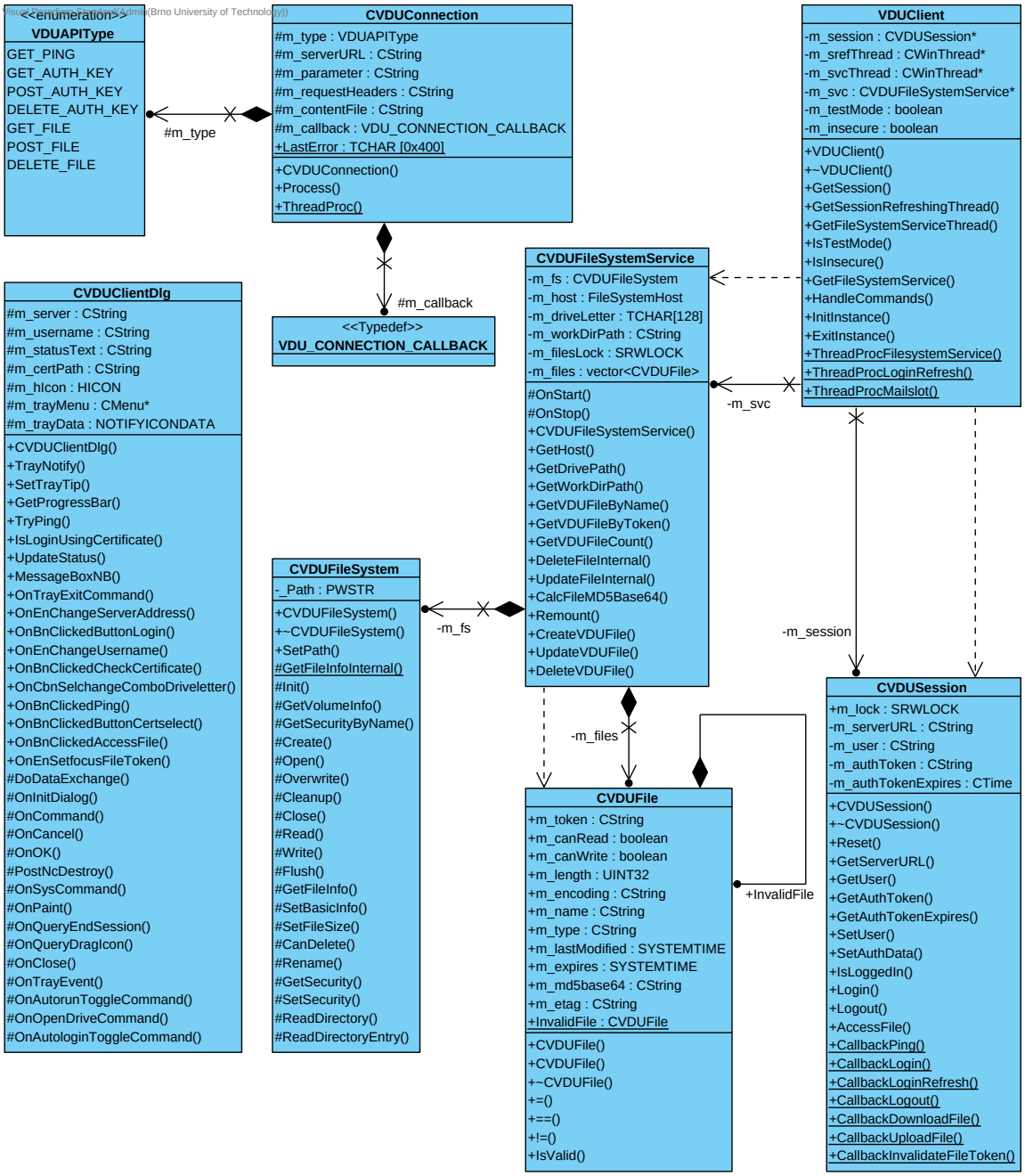

Figure 4.1: UML class diagram of the VDU Client application.

Each class has a name that implies its purpose and responsibility. All classes rely on each other to exist and have to be instantiated for the VDU client to function, except the dialog handling class, `CVDUClientDlg`, which is not required to be instantiated when the application is being tested. Running tests on the application is further described in Chapter 6.

4.1.2 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; the persistent state of the application, stored in the Windows registry

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 the 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.

Motivation

Creating a virtual file system gives the freedom to portray any data as files. This has led me to attempt to store the files using just the system's Random-Access Memory (RAM) provided to the application. While the speed and accessibility of RAM make this idea seem plausible, in practice, it did not work well for the following 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

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. The application's configuration settings can be simplified into a simple pair of `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 persistent state of the application. Table 4.1 lists out the concrete settings stored inside the registry by the application. These

values are stored under the `HKEY_CURRENT_USER` key. This guarantees that different Windows users on the same machine will have their own configurations of 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 4.1: The concrete settings stored by the application in the Windows registry.

4.2 User interface

The user interface includes every visual element of the application and all 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* or cases. Figure 4.2 displays all use cases of the interface.

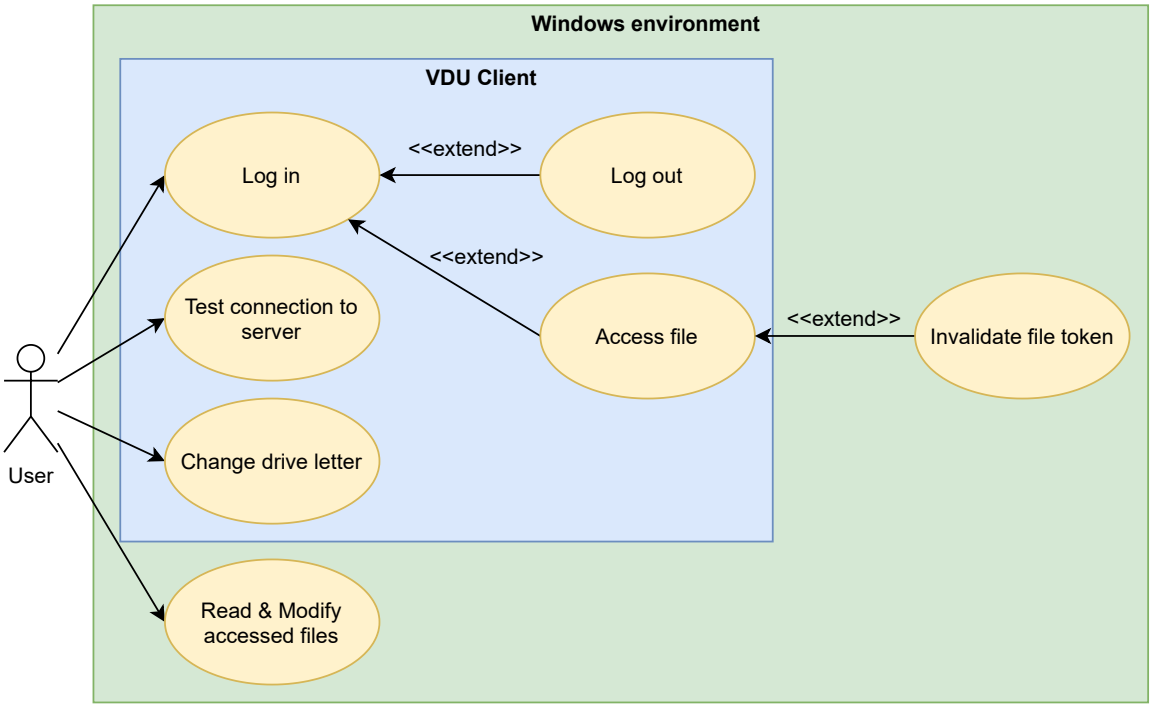

Figure 4.2: Use case diagram from the view point of an user of the VDU Client system and the Windows environment it resides in.

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

4.2.1 Integration into the Windows environment

Not all user 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. Additionally, the Windows environment provides useful functionality, which could provide a quality-of-life improvement for the user experience.

Windows File Explorer

The Windows File Explorer (Explorer), a built-in tool for browsing files in Windows, has its own 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 4.3.

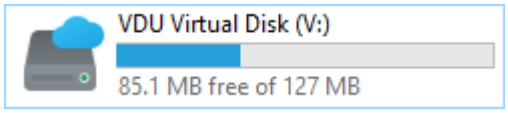

Figure 4.3: The VDU Virtual Disk as shown in the Windows File Explorer.

Using the Explorer to provide the functionality of file access-related actions is an intuitive way of integrating with the Windows desktop environment. Figure 4.4 shows an example for browsing VDU files using the Explorer.

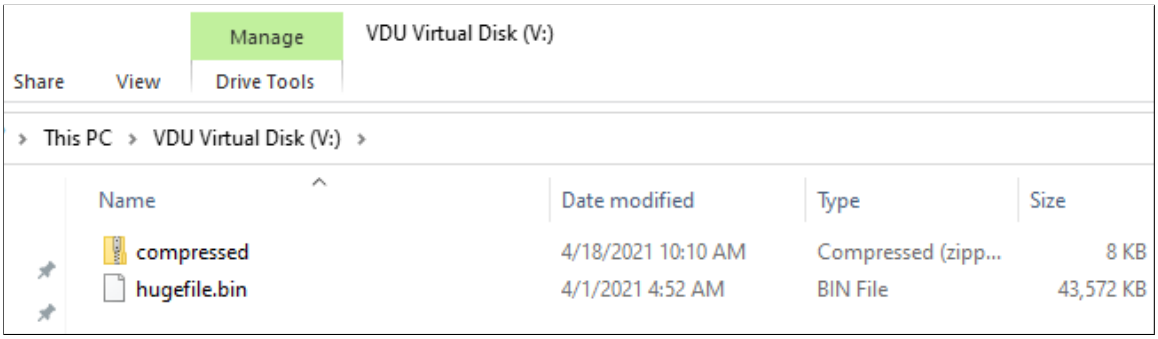

Figure 4.4: Browsing files in the VDU Virtual Disk using the Windows File Explorer.

The VDU API provides an endpoint for file token invalidation, which has led me to make a design choice to use the delete feature of the file system as a trigger for invalidation. The point is re-purposing the file deletion feature to invalidate the file's token and only delete the local file if the server allows so. Thanks to this approach, the number of actions that the VDU client has to contain in its own user interface is reduced further. In conclusion, by using built-in features of existing applications in the Windows environment, specifically the Explorer, I was able to simplify the VDU client's user interface while keeping its core functionality unchanged.

Single instance

To make the user experience better, the application will only run under a single instance. Any additional instances of the application will exit immediately, leaving only one process of the application running in the system – the main process. To improve the usability of the application, any additional instances will, before exiting, pass their command lines to

the main process. This design choice gives the freedom to control the running application via the command line.

Protocol association

For easier access to VDU files, the application creates a new URL[2] protocol. Other applications, such as an internet browser, can command the VDU Client application to access a file by the token provided in the URL path. The protocol has the URL format of `vdu://{token}`, where the `token` represents the file access token of the file, which the application should access. This allows many applications, and most importantly, websites, to include hyperlinks that use the VDU URL protocol to access files, as displayed in Listing 4.5.

Figure 4.5: An example of a VDU URL protocol hyperlink, displayed on a website using an internet browser, which can be clicked to access a file.

4.2.2 Dialog design

After narrowing down all user actions, as shown in the previous subsection, the next step is to design the actual VDU client interface.

Considering the little functionality required, I decided to design a simple *Extended Dialog Window* (dialog). The dialog, seen in Figure 4.6, 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 the 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.

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.

[2]Uniform Resource Identifier

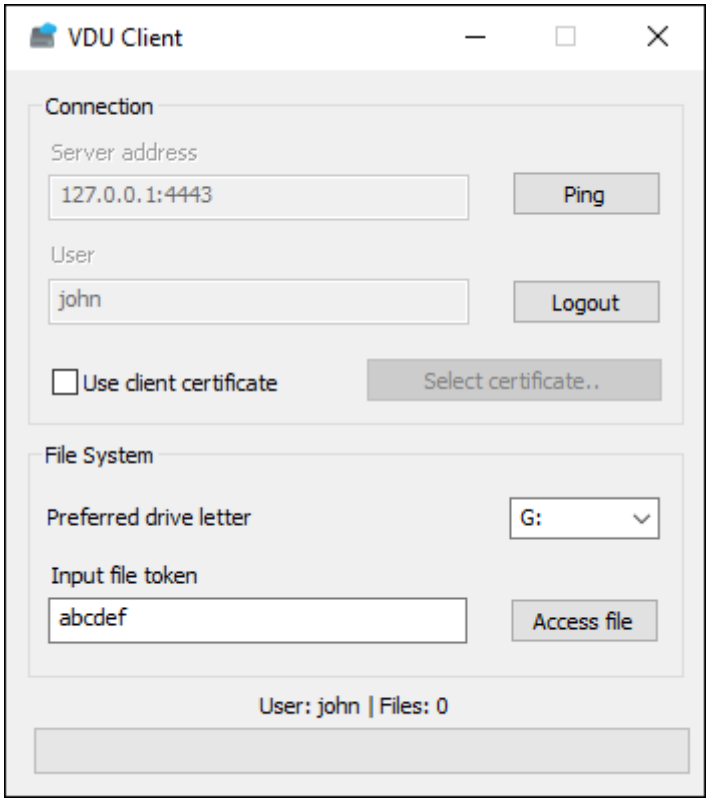

Figure 4.6: User interface of the VDU Client application.

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 the currently logged-in user.

4.2.3 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. Upon closing or minimizing the dialog, the application stays running in the background, signified by the icon. The user can restore the dialog window by simply clicking on the VDU Client icon, provided by Icons8[3], displayed in Figure 4.7, in the tray area.

[3] https://icons8.com/

Figure 4.7: Cloud Storage Icon, used as the main icon for VDU client application. Source:[9].

Furthermore, removing the ability to close and exit the application from the dialog window has moved this responsibility to the tray icon. I designed a simple tray menu to fix this problem, depicted in Figure 4.8, which becomes active after right-clicking the icon. The *Exit* option in this menu, shown in the listing, is how the user is supposed to quit using the application.

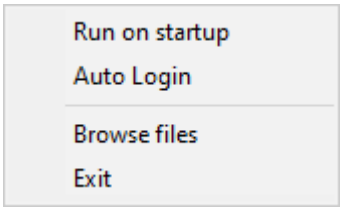

Figure 4.8: Tray menu of the VDU Client application.

Windows allows applications that put icons in the tray area to display a small text message while hovering the icon – a *tray tip*. I utilized this tray tip to display a compact text version of the data from the status section of the dialog, as displayed in Figure 4.9.

Figure 4.9: Example of a tray tip, displaying the state of the application.

4.2.4 Responsiveness and notifications

A good application needs to be responsive and notify the user about important events. Every action directly taken by the user should have a visual response. Given the specifications, the VDU client application will communicate with a server over a network connection. Whether the server resides in a local network or on the internet, it is safe to assume that the application will not finish an action instantaneously. This means a 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 all actions. For the VDU client dialog window, an instant response consists of enabling or disabling the related child windows. For example, an instant response to clicking the *Ping* button, as displayed on the

user interface in Figure 4.6, 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 4.10.

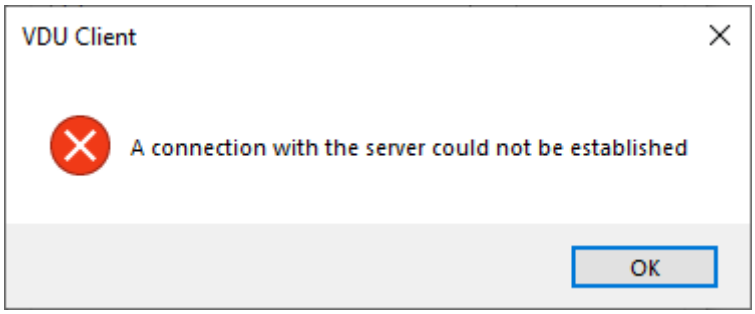

Figure 4.10: An example message box dialog, as displayed after the application fails to connect to a server.

Windows notifications

For less important and rather informative actions, a Windows notification shows up as a response. Such a notification, displayed in Figure 4.11, 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 application that caused a different application to open.

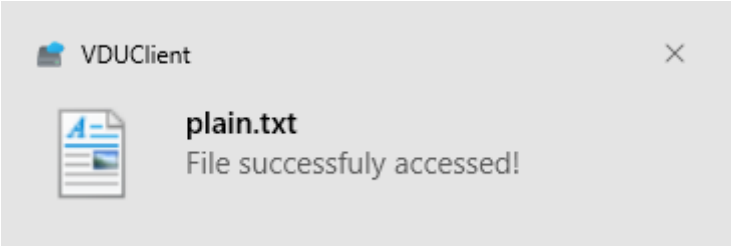

Figure 4.11: An example of Windows notification, as displayed in the bottom right corner of the screen after successfully obtaining access to a file.

4.3 Automated tests

By default, the application can only be controlled by the user using its user interface. This makes it difficult to test the application automatically, as issuing commands to the user interface is not practical. A much better solution for this problem is to create a special mode in which the application can be launched - the *test mode*. Furthermore, this section

designs a simulated VDU server since the automated tests of the application should ideally be performed without the presence of a production VDU server.

4.3.1 Test mode

The application can be launched into the test mode with the `-testmode` launch option. In the test mode, the dialog window of the application is not created, and the application starts executing *test instructions* in the background.

Instructions

A test instruction is an additional, repeatable launch option, which simulates a user action with the application. Each instruction represents a single unit of functionality that can be tested. The list of available test instructions is the following:

- `-server [address]` – Set the server address
- `-user [name]` – Login as user 'name'
- `-logout` - Invalidate authentication token
- `-accessfile [token]` – Access a file by token
- `-accessnetfile [url]` - Access a file by token inside the VDU URL protocol
- `-deletefile [token]` – Invalidate a file by token
- `-rename [token] [name]` – Rename an accessed file, recognized by a token, to a new filename
- `-write [token] [text]` – Write a Unicode string at the beginning of the accessed file, recognized by a token, and upload it to the server
- `-read [token] [cmpText]` – Read a Unicode string of the length of the input text from the beginning of the accessed file recognized by token, and compare it to the input text

Instructions do not modify the persistent state of the application outside of the test mode. All user settings will remain intact after using the application in test mode.

4.3.2 Simulated server

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 over the production server:

- *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 by the speed of the local network

Structure

Considering the simplistic nature of the server's requirements, I designed the simulated server as a simple script using Python[4]. The server consists of a single file, which upon execution, starts an HTTPS server, accepting connections over a secure channel with all endpoints available to respond to requests.

Simulating endpoints

The specification of the VDU server API provides an overview of the design of all available endpoints. The simulated server attempts to replicate the expected functionality of those endpoints to represent a production server to a reasonable extent, the validity of which can be confirmed by automated tests.

4.3.3 Testing script

In addition to the simulated server, and as an explicit requirement of the specification, I designed a Python testing script. This script consists of a single file, `test.py`, which uses both the application executable and the simulated server script. Upon execution, the script will perform all included tests and show the results of each test.

Execution

The script will start the execution by starting the simulated VDU server. Once the server is started, the testing script will start the VDU Client application in the test mode, with the respective test instructions input. The testing script waits until the application process terminates due to the testmode parameter and compares the process exit code. The exit code determines whether the test has been successfully performed.

Testing process

The testing script should contain enough different tests to validate the functionality of the individual components of both the application and the simulated VDU server. The tests included in the testing script should assure the correct functionality of the features, which will be used by real users and will have a critical impact on the application's usability. The included tests should verify the following actions:

- Connecting to a server
- Authentication
- Invalidating authentication
- Accesing a file
- Invalidating a file token

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

- Renaming a file
- Writing to a file
- Reading from a file

Chapter 5

Implementation

This chapter covers the exact internal implementation of key elements of the VDU client, as described in the previous chapter, which covered the application's overall design. The implementation of the application was done using Visual Studio Community 2019, an IDE[1] described in Section 2.1.1. Listings in the following sections make implicit use of the following macros:

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

5.1 Internal components

The back-end implementation of the application's internal components corresponds to the class structure designed in Chapter 4. This section covers the details of important parts of the implementation related to the required application functionality.

5.1.1 Connection and session

The VDU connection is an extended wrapper of a regular HTTP connection to a VDU server, which handles 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 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, the client must send each request separately from one another. Implementing requests regularly creates too much redundant code if the application needs to receive the response as well.

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

[1] Integrated Development Environment

request to the VDU server can be simplified into creating a connection object, as shown in Listing 5.1.

```
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 5.1: Example of instantiating a VDU connection wrapper class.

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.

I solved this issue by creating a permanent worker thread on the application's 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. Instead of calculating the exact time a thread should sleep, the reasoning behind the one-second interval 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.

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 when waiting for the 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 its execution at the beginning of a static function, representing the thread procedure, `CVDUConnection::ThreadProc`, which processes the connection and deletes the object from memory afterward. Creating a thread using `AfxBeginThread`, as shown in Listing 5.2, is a non-blocking operation, ensuring 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 5.2: Creating a new thread to process a connecton which sends a login request to the server.

Thread synchronization

In a scenario where one or more worker threads are processing connections simultaneously, all threads of the program are subject to a *data race*. The data race occurs due to the

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 its 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 5.3. 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.

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

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

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

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

Listing 5.3: Example of a critical section implementation using an SRW lock.

Callback functions

To handle the example login request results demonstrated in Listing 5.2, the caller can specify a *callback* function. 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 session data* – To prevent data racing The r
- *Return value is thread exit code* – For synchronous operations

Additionally, a callback function must follow the prototype, declared in Listing 5.4. The parameter is the received response from the server in the form of a `CHttpFile` object. The return value, as specified above, will be used as a return code for the running thread. This assures that another synchronization object can check the results of this callback operation after the running thread terminates.

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

Listing 5.4: The prototype of a VDU callback function.

5.1.2 Virtual file system

The VDU virtual file system is originally based on the *passthrough-cpp*[3] example file system made by *Bill Zissimopoulos*, available from [5]. The original example file system implemented the functionality of accessing a given directory path via the virtual drive directly,

[2] Slim Reader/Writer lock

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

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 similar sense. Basing the VDU virtual file system on that system 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 [38], 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

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

Read-only files

VDU files can have a property, which disallows 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 the server will deny.

A decent approach to this issue is to disallow programs from acquiring a file's 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 and in the `Create` function to prevent modifying the file by replacing it. Listing 5.5 shows the implementation of the access right check.

```
1  NTSTATUS CVDUFileSystem::Open(PWSTR FileName, UINT32 CreateOptions, UINT32
       GrantedAccess, PVOID* PFileNode, PVOID* PFileDesc, OpenFileInfo* OpenFileInfo)
2  {
3      ...
4      //Find requested file
5      CVDUFile vdufile =
           APP->GetFileSystemService()->GetVDUFileByName(PathFindFileName(FileName));
6      ...
7      //If the file is a VDU file
8      if (vdufile.IsValid())
9      {
10         //If the file is read-only, check writing rights
11         if (!vdufile.m_canWrite &&
12                (GrantedAccess & GENERIC_WRITE ||
13                 GrantedAccess & FILE_APPEND_DATA ||
14                 GrantedAccess & FILE_WRITE_DATA))
15         {
16             //Return a descriptive NTSTATUS
17             return STATUS_MARKED_TO_DISALLOW_WRITES;
18         }
19     }
20     ...
21 }
```

Listing 5.5: Implementation of the read-only check for files, to disallow creating a handle with write access.

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 response headers and starts downloading the file. The file is first downloaded into the current Windows user's temporary directory with a temporary name prefixed with the 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 successfully, without modification.

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, and the file is registered. It is moved into the applications work directory,

available to be accessed by the user through the virtual file system. Calculating the MD5 hash of a file is implemented according to [11] using the cryptographic functions of Windows API.

Uploading files

Normally, local VDU files must be manually uploaded to the VDU server every time a significant 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 repetitive tasks.

To automate this process, the application will only upload the local VDU files present in the virtual file system to the server if a change to the file's contents is detected. If a change is detected, the file system service will upload the file in the background without the need for 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. The application responds by removing the file locally and notifying the user about this occurrence.

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 system implementation. An application can change a file 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 virtual function `Rename` 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 that handles it. Usually, an application that replaces files creates a temporary file, writes new contents into that temporary file, and renames the temporary file to the original file name. It makes use of the renaming functionality. To differentiate user-triggered renaming and renaming caused by the application's overhead, I use the `ReplaceIfExists` parameter of the `Rename` virtual function of the virtual file system. This parameter is `False` for renaming due to overhead and `True` for user-triggered renaming by Windows File Explorer. While the overhead renaming could be used as a detection vector for file changes, I decided to use a more efficient approach.

Detecting *direct modification* of a file stems from an object, which is being used while modifying a file – *a handle*. No application wants to keep a handle to a file for too long, as it would potentially prevent other applications from accessing this file. That is why

intercepting the end of the modification process when an application is closing a handle to a file is the best spot for detection. The virtual function `Close` provides the handle, which is about to be closed via the `FileDesc0` parameter. To determine whether this file is one of the VDU files, the `GetFinalPathNameByHandle`[4] function provides the file name of the file this handle belongs to. The application can compare it 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 intends only to 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, the file system needs to figure out whether a handle has write access to the file it belongs to. The internal Windows API function `NtQueryObject` allows querying the `GrantedAccess` of a handle object, as specified by [17]. Unlike other Windows API functions, it is only accessible via a dynamic link. The implementation of acquiring the link to `NtQueryObject` and using it to identify handles with write access is shown in Listing 5.6. Thanks to this approach, the identifying process is more effective, as only handles with write access are considered for a file modification check.

Version control

Each VDU file that is accessed has its version controlled by the VDU server. The version string is accessible in the response header as the `ETag` header. The exact content of the version string is not specified.

To supplement this fact, I implemented the version as simply following the orders of the server, given that the client application has no control over the file version - it can only respect it. When a file is modified, the file is uploaded to the VDU server. If the server accepts the updated file, it will return a new version in the `ETag` header of the response. This is updated internally so that any further changes will be done with respect to the current version of the file. Contrary to how the thesis specification purports the application version control, the application can at its best follow the server's instructions about the version of a file.

Invalidating file tokens

Invalidating a file token when the user no longer needs a file to be accessible was designed to be done by repurposing the delete feature of Windows File Explorer. However, the virtual file system provides no virtual function, called when a file is being deleted.

After some testing, I found out that the delete feature of Windows File Explorer is actually implemented as simply opening the file with the `FILE_FLAG_DELETE_ON_CLOSE` flag and closing the handle right after. In conclusion, to repurpose the delete function, I intercept the `Open` virtual function of the virtual file system and check the `CreateOptions` parameter for this flag.

Custom drive icon and label

For better visual clarity and easy recognition of the difference between a physical 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,

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

```
VOID CVDUFileSystem::Close(PVOID FileNode, PVOID FileDesc0)
{
    ...
    VdufsFileDesc* FileDesc = (VdufsFileDesc*)FileDesc0;

    //Prototype from documentation
    typedef NTSTATUS (NTAPI* NtQueryObjectFn)(HANDLE Handle,
        OBJECT_INFORMATION_CLASS ObjectInformationClass, PVOID ObjectInformation,
        ULONG ObjectInformationLength, PULONG ReturnLength);

    //Get the function dynamically from ntdll
    static NtQueryObjectFn NtQueryObject = NULL;
    if (!NtQueryObject)
    {
        if (HMODULE ntdll = LoadLibrary(_T("ntdll.dll")))
            NtQueryObject = (NtQueryObjectFn) GetProcAddress(ntdll,
                "NtQueryObject");
    }

    //Assume the handle has writing rights
    BOOL handleHasWriteRights = TRUE;
    PUBLIC_OBJECT_BASIC_INFORMATION pobi;

    //Query the handle object's basic information
    if (NtQueryObject != NULL && NT_SUCCESS(
        NtQueryObject(FileDesc->Handle, ObjectBasicInformation, &pobi,
            sizeof(pobi), 0)))
    {
        ACCESS_MASK GrantedAccess = pobi.GrantedAccess;

        //Queried successfully, assume no write rights
        handleHasWriteRights = FALSE;

        //Check handle's access for write flags
        if (GrantedAccess & GENERIC_WRITE ||
            GrantedAccess & FILE_APPEND_DATA ||
            GrantedAccess & FILE_WRITE_DATA)
        {
            handleHasWriteRights = TRUE;
        }
    }
    ...
}
```

Listing 5.6: Implementation of accessing the NtQueryObject function of ntdll.dll and using it to identify wheter a handle, that is about to be closed, has access to writing to the file it belongs to.

displayed in Figure 4.7, and include a descriptive drive label to match. To change a drive 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 the '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. According to [32], for Windows versions other than *Windows 2000*, which is this application's case, the key is located under the `HKEY_LOCAL_MACHINE` key. This would change the drive icon for every Windows user and requires administrator permissions to change.

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 [26], the `HKEY_CLASSES_ROOT` key will be replaced to the `HKEY_CURRENT_USER\Software\Classes` key, if the intent is to write only to the current Windows user's settings. That is exactly the application's intent, as it requires no additional rights. An example of an implementation using a modified registry path to enable a custom icon is shown in Listing 5.7.

```
1  NTSTATUS CVDUFileSystemService::Remount(CString DriveLetter)
2  {
3      ...
4      CRegKey key;
5      if (key.Create(HKEY_CURRENT_USER,
           _T("SOFTWARE\\Classes\\Applications\\Explorer.exe\\Drives\\") +
           CString(m_driveLetter[0]) + _T("\\DefaultIcon")) == ERROR_SUCCESS)
6      {
7        //Acquire the executable path, containing the icon
8        CString moduleFilePath;
9        AfxGetModuleFileName(NULL, moduleFilePath);
10       //Select the first icon, set it to the default value
11       key.SetStringValue(NULL, _T("\"") + moduleFilePath + _T("\",0"));
12       key.Close();
13     }
14     ...
15 }
```

Listing 5.7: Implementation of applying a custom drive icon to a drive, noted by its drive letter, when re-mounting the virtual file system.

5.2 User interface

The front end of the application is the user interface, as it was designed in Chapter 4, 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, and the application window, unless specified otherwise.

5.2.1 Dialog window

The dialog window was implemented using MFC[5] in Visual Studio, which allows creating dialog interfaces in a schematic-like format, as displayed in Figure 5.1. Each designed window serves as a template for programmatically created windows of the same type and

[5]Microsoft Foundation Class library

has a unique *name*. When creating a window, the developer can specify a template to guide the design of this window. As shown in Listing 5.8, the `Create` function of `CDialogEx` is using the designed window template specified as the first parameter.

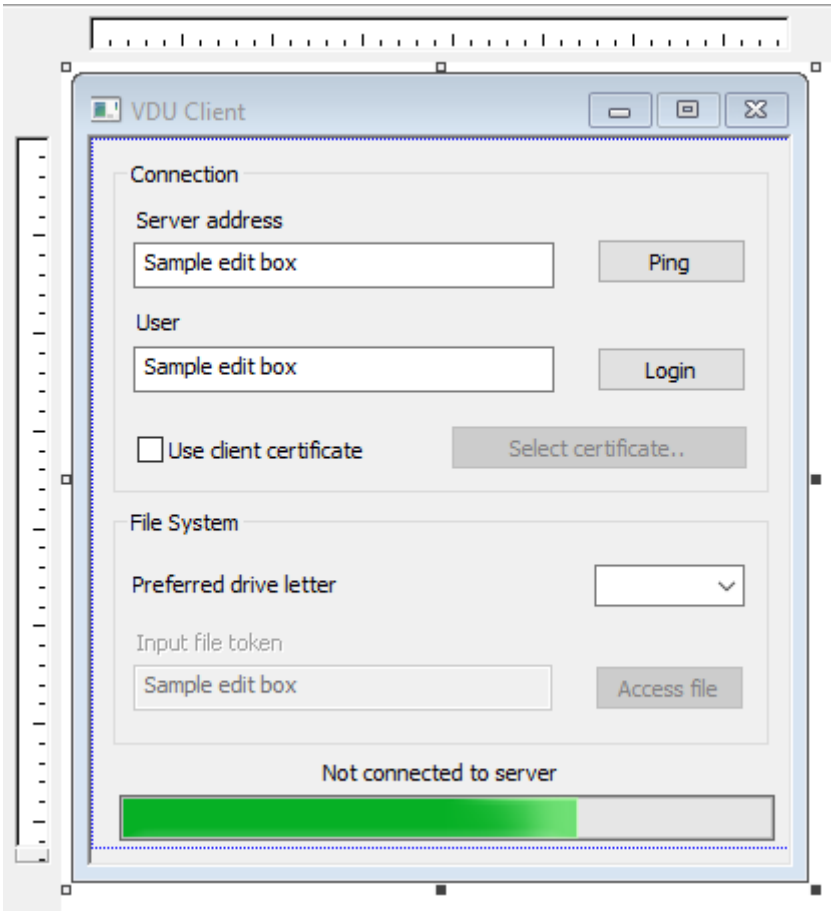

Figure 5.1: The VDU Client dialog window user interface; created in the Visual Studio Dialog Editor.

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

Listing 5.8: 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 called when a specific message is received - an *event handler*. For each child window of the dialog, including itself, I implement one or more event handlers to assure functionality of the interface. An example of an event handler and its mapping to a message and implementation is displayed in Listing 5.9.

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

//The message handler function
void CVDUClientDlg::OnBnClickedPing()
{
    TryPing();
}
```

Listing 5.9: 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 the 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.

To make it seem like the window gets hidden to the tray, I override the system command `SC_CLOSE`[7] to minimize and hide the main window, as shown in Listing 5.10.

```
//Overriding the OnSysCommand virtual function
void CVDUClientDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFF0) == SC_CLOSE)
    {
        //Hide to tray
        ShowWindow(SW_MINIMIZE);
        ShowWindow(SW_HIDE);
        return;
    }

    //Call parent's default implementation
    CDialogEx::OnSysCommand(nID, lParam);
}
```

Listing 5.10: Overriding system close command to hide the dialog window

Message box

A message box is a small dialog that informs the user about run-time errors or important events. When a message box is active, its presence blocks the thread which creates it. This ties closely to the thread synchronization mentioned in the previous section and could lead to synchronization delays. It could even cause a deadlock if a thread does not release the synchronization lock.

To prevent this while still using message boxes, each message box is created in a separate new worker thread. The calling thread creates a structure consisting of the parameters of

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

[7] https://docs.microsoft.com/en-us/windows/win32/menurc/wm-syscommand

the message box and passes it to the new thread. This approach avoids any potential delays in thread synchronization.

5.2.2 Windows environment

This subsection covers features that tie closely to the interface of the Windows environment. The specification does not explicitly require their presence. Instead, they serve as quality of life improvements of the user experience.

Browsing and opening files

After successfully accessing a VDU file by its token, the file is created locally and available on the virtual file system's virtual drive. It might not be clear how to access this place or run the newly accessed file simply for some users.

After a file is accessed, the application automatically starts the program associated with this file's type. This is done by using the `ShellExecute`[8] function to open the file at its location. I also used this function to open the Windows File Explorer at the path of the virtual drive, which is accessible through the tray popup menu entry – *Browse files*. The usage of `ShellExecute` in the implementation of these features is shown in Listing 5.11.

```
1 //Browse files inside the virtual drive path
2 ShellExecute(WND->GetSafeHwnd(), _T("explore"),
      APP->GetFileSystemService()->GetDrivePath(), NULL, NULL, SW_SHOWNORMAL);
3
4 //Open VDU file ‘plain.txt‘ with the assigned program
5 ShellExecute(WND->GetSafeHwnd(), _T("open"),
      APP->GetFileSystemService()->GetDrivePath() + _T("plain.txt"), NULL, NULL,
      SW_SHOWNORMAL);
```

Listing 5.11: Examples of using ShellExecute to open and browse files

Running on startup

To make Windows automatically run programs after the current Windows user logs in, as [35] describes, the path to the application executable is set as a string value inside `CurrentVersion\Run` subkey of the `HKEY_CURRENT_USER` key. With this registry value set up, the application will be launched each time the Windows user logs in. However, the user might want this feature enabled at all times.

To solve this issue, I wanted to replicate how the Windows Task Manager disabled and enabled the startup applications and use this feature in the application. As there is no official documentation for the Task Manager, I took inspiration from the notes of *René Nyffenegger*, available at [21], where he describes the registry subkey `StartupApproved\Run`. The values in this subkey describe whether an application is enabled or disabled from starting up – it overrides the settings of the default startup registry path. However, the notes contain incomplete information about the startup entry structure.

According to my testing, from the first 4 bytes of the binary value, only the first bit is used to determine whether the entry is disabled. The second bit always gets set when the *Startup* tab of the Task Manager is first displayed. This bit does not get set again until the

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

Task Manager is restarted – it could have been an oversight, as it does not serve a purpose related to application startup.

I created a simple structure, as shown in Listing 5.12. which represents the 12 bytes saved in the registry. The VDU Client reads and writes this structure to the registry.

```
typedef struct StartupApprovedEntry_t
{
    enum FlagBits
    {
        DISABLED = (1 << 0), //The application will not start
        TASKMGR_VIEWED = (1 << 1), //The entry was viewed in the Task Manager
    };

    DWORD flags; //Contains flag bits
    FILETIME disabledTime; //Time when the entry was disabled
} StartupApprovedEntry;
```

Listing 5.12: The internally implemented structure of an entry of a startup-approved application of modern Windows.

Single instance

By default, upon starting another instance of the application, it will create a new dialog window and attempt to create the virtual file system drive. This could lead to issues, as the previously created instance has already created the drive, or confusion related to the number of dialog windows and tray icons.

To achieve this, I used a mailslot. The first, main instance of the application creates a mailslot with a predefined name, and keeps its handle until it exits. If another instance of the application attempts to create the same mailslot, the creation will fail – the application process can recognize whether it is the main process or not.

To make a better use of the mailslot, the main application process will create a worker thread, which continuously attempts to read new messages from the mailslot. Every read message will be passed to the `VDUClient::HandleCommands` function, which parses and handles the command line synchronously. To achieve this, the application process, which is not the main one, opens the mailslot and writes its own command line as a message before exiting. This allows for many processes of the application to control a single instance instead of instantiating themselves.

Protocol association

According to [15], to associate the application with the `vdu` URL protocol, the application should register this protocol as one of the root classes. To register the protocol only for the current Windows user, the registry key used is under the `HKEY_CURRENT_USER` key, instead of the `HKEY_CLASSES_ROOT` key, as mentioned in previous sections. When the protocol is triggered, a new instance of the application is launched with the command to access the requested file by the token in the URL path.

The implementation of this ties to the single instance feature of the application. The newly created instance, triggered by launching the VDU URL protocol, will write the ‘-accessnetfile‘ command into the mailslot. This will cause the main process to read the new message and execute the command, leading to seamless access to the requested file via the currently running application.

5.3 Automated tests

This section intends to implement the automated tests designed in the previous chapter. All automated tests are implemented using the Python programming language with a single script. The tests are run on the application in the test mode, which causes the application to validate the results of each instruction and exit with the respective exit code. All tests assume the persistent state of the simulated VDU server.

5.3.1 Test mode

The test mode of the application is implemented as a simple variable, indirectly checked in different parts of the application. In the test mode, the application executes the instructions in its own command line. While it does create a mailslot, to prevent from creating more instances of the application, it does not read from it to prevent it from tampering with the currently running test.

Executing instructions

Instructions of the application are contained in the applications command line. The function `VDUClient::HandleCommands` parses, and executes every instruction that is matched with the instruction set and ignores others. While the application is running in the test mode, the instructions are always executed synchronously in the main thread. This is possible thanks to the dialong window not being active, and it is different from the regular mode, where the instructions are still run synchronously, but in a worker thread.

Error handling

If an instruction communicates with the server during its execution, it awaits the response from the server in the same thread – it is blocking. After evaluating the response, if the action was not successful, the application immediately exits with the respective exit code. Otherwise, the application exits after the last instruction is executed with the exit code `EXIT_SUCCESS`.

5.3.2 Simulated server

The simulated VDU server is implemented as a single Python script. This script creates an HTTP server, to which it assigns a custom `VDUHTTPRequestHandler` objected, which is an extended version of the `BaseHTTPRequestHandler`[9] class. This class implements all endpoints of the VDU server API. The server is bound to all interfaces and started on port `4443` by default.

Support for HTTPS

The support for the HTTPS protocol was added according to [2], by generating a server certificate using OpenSSL[10]. This certificate is then used to wrap the HTTP server's socket, creating an HTTP server with HTTPS support. The certificate is not a valid certificate signed by an authority, and should only be used for testing.

[9] https://docs.python.org/3/library/http.server.html
[10] https://www.openssl.org/

Predefined data

The server includes predefined simulated data for the purposes of testing. This data includes a few example users and examples of always active file tokens for accessible VDU files. This helps to create a good environment for testing, as tests can be created to assume the persistent state of the simulated server, where known users and file tokens are available.

Endpoint simulation

The simulated server attempts to replicate the functionality of the endpoints of the production server using the programming constructs of Python and its default libraries. Authentication related endpoints are simulated using an array to manage the generated authentication tokens. The file related endpoints use the operating system library[11] to gain information about the VDU files, such as their size, access rights, file name, etc.

The point is not to accurately simulate how a production server would handle the request internally, but rather simulate accurate responses to client requests. For this reason, some endpoints have simplified internal functionality. For example, the simulated `DELETE /file/` endpoint does not actually invalidate file tokens on success.

Version control

All predefined files have a certain starting version. This version is incremented each time a file is successfully uploaded to the server. This creates a simulation of how a simple version control system would work – the previous versions are not backed up anywhere, as they would be on a production server.

Failures

The specification of the VDU API includes, for some endpoints, status codes for internal failures such as read timeouts. To simulate internal read timeouts, I added a probability of failure setting, which affects file-related endpoints. This setting should remain 0 while using the server to perform tests.

5.3.3 Testing script

The testing script is implemented inside a single Python script. It contains the definitions of automated tests to be run on the application predefined directly in the code, inside an array. The array can be extended by creating new tests, which will be run along with the predefined ones.

Performing tests

The tests are performed using the subprocess[12] Python library. This library allows the script to create new process based on the input command line. To run a single test, the script performs the following steps:

1. *Start the simulated VDU server*
2. *Start the VDU Client in test mode with test instructions*

[11] https://docs.python.org/3/library/os.html
[12] https://docs.python.org/3/library/subprocess.html

3. *Wait for the VDU Client to terminate*
4. *Terminate the simulated VDU server*
5. *Compare the exit code of VDU Client to the expected code*

These steps are done for every test in the array. Afterwards, the script prints the results of the testing to the standard output and awaits user input before exiting.

Chapter 6

Testing

This chapter

6.1 Performing automated tests

The automated tests for the application, as designed and implemented in the previous chapters, can be performed using the testing script. The results of these tests will then signify whether the application succeeds in conforming to the specified requirements, whether the application functions correctly, and whether the simulated server's behaviour matches the expected behaviour of the production server.

6.1.1 Windows 10

Test

```
[02:01:52] [Test] OK [server_bad] 1
[02:01:53] [Test] OK [login_ok] 0
[02:01:53] [Test] OK [login_bad] 1
[02:01:54] [Test] OK [logout_ok] 0
[02:01:54] [Test] OK [logout_bad] 1
[02:01:55] [Test] OK [nologin] 1
[02:01:56] [Test] OK [del_ne] 1
[02:01:57] [Test] OK [file] 0
[02:01:57] [Test] OK [nofile] 1
[02:01:59] [Test] OK [thetwotime] 0
[02:02:00] [Test] OK [tworeqs] 1
[02:02:01] [Test] OK [allfiles] 0
[02:02:02] [Test] OK [rename_bf] 0
[02:02:03] [Test] OK [rename_ne] 1
[02:02:04] [Test] OK [write_ok] 0
[02:02:05] [Test] OK [write_ne] 1
[02:02:05] [Test] OK [write_multi] 0
[02:02:06] [Test] OK [read_ok] 0
[02:02:07] [Test] OK [read_bad] 1
[02:02:08] [Test] OK [read_two] 0
[02:02:08] ====================
[02:02:08] [Test] Passed 20/20 tests
```

Listing 6.1: Output of the testing script on a Windows 10 machine, after running all 20 predefined tests.

6.1.2 Results

6.2 Backward compatibility

6.2.1 Windows 8

Test name	Exit code	Success

Table 6.1: The results on Windows 8.

6.2.2 Windows 7

6.2.3 Conclusion

Chapter 7

Conclusion

This thesis aimed to design and implement a client-side application for Microsoft Windows, which would provide secure access to remote files on the VDU server, based on the specification and requirements. To achieve this, this thesis introduces the development of applications for Microsoft Windows, overviewed important technologies and libraries. Furthermore, it introduced file systems and their abstractions, virtual file systems. It listed and reviewed multiple options of the third-party file system software, from which it picked the best one and built upon it. The server requirements were analyzed and formalized, and the application was designed in detail, along with the automated test and a simulated VDU server. The application, the simulated server, and automated tests were implemented and tested both manually and automatically. Additionally, backward compatibility support was tested to prove the versatility and stability of the project. The resulting software is published as open-source on GitHub[1], as required by the specification. It is available to be extended and developed further. It could be improved in the future by extending the simulated server functionality to be more precise to the production server, to tie closer to the VDU web user interface where the file tokens are generated, or to add more quality of life options according to the requirements of users.

[1] https://github.com/coolguy124/vduclient

Bibliography

[1] 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.

[2] Dergachev, A. *Simple-https-server.py* [online]. 2021 [cit. 2020-04-27]. Available at: https://gist.github.com/dergachev/7028596.

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

[4] 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.

[5] *Billziss-gh/winfsp: Windows File System Proxy - FUSE for Windows* [online]. GitHub, 2021 [cit. 2020-04-20]. Available at: https://github.com/billziss-gh/winfsp.

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

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

[8] 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/.

[9] *Cloud Storage Icon – Free Download, PNG and Vector* [online]. Icons8, 2021 [cit. 2020-04-20]. Available at: https://icons8.com/icon/r8kHwiV6nVEd/cloud-storage.

[10] 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.

[11] lastnameholiu, Sharkey, K., Coulter, D. et al. *Example C Program: Creating an MD5 Hash from File Content - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-04-25]. Available at: https://docs.microsoft.com/en-us/windows/win32/seccrypto/example-c-program--creating-an-md-5-hash-from-file-content.

[12] 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.

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

[14] Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ: Prentice Hall, 2009. 138-140 p. ISBN 0-13-235088-2.

[15] *Registering an Application to a URI Scheme (Windows) | Microsoft Docs* [online]. Microsoft, 2016 [cit. 2020-05-04]. Available at: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa767914(v=vs.85).

[16] *Local File Systems (Windows) | Microsoft Docs* [online]. Microsoft, 2018 [cit. 2020-03-10]. Available at: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/aa364407(v=vs.85).

[17] *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.

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

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

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

[21] Nyffenegger, R. *Registry: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\StartupApproved\Run* [online]. 2021 [cit. 2020-04-20]. Available at: https://renenyffenegger.ch/notes/Windows/registry/tree/HKEY_CURRENT_USER/Software/Microsoft/Windows/CurrentVersion/Explorer/StartupApproved/Run/index.

[22] 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-.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Schofield, M., Sharkey, K., Coulter, D. et al. *Predefined Keys - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-04-22]. Available at: https://docs.microsoft.com/en-us/windows/win32/sysinfo/predefined-keys.

[29] 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.

[30] 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.

[31] Schofield, M., Sharkey, K., Coulter, D. et al. *About Mailslots - Win32 apps | Microsoft Docs* [online]. 2021 [cit. 2020-04-29]. Available at: https://docs.microsoft.com/en-us/windows/win32/ipc/about-mailslots.

[32] 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.

[33] 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.

[34] 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.

[35] Schofield, M., Sharkey, K. and Satran, M. *Run and RunOnce Registry Keys - Win32 apps | Microsoft Docs* [online]. 2018 [cit. 2020-04-20]. Available at: https://docs.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys.

[36] *Native API vs FUSE · WinFsp* [online]. secfs, 2021 [cit. 2020-04-20]. Available at: http://www.secfs.net/winfsp/doc/Native-API-vs-FUSE/.

[37] *WinFsp* [online]. secfs, 2021 [cit. 2020-04-20]. Available at: http://www.secfs.net/winfsp/.

[38] *WinFsp Tutorial · WinFsp* [online]. secfs, 2021 [cit. 2020-04-20]. Available at: http://www.secfs.net/winfsp/doc/WinFsp-Tutorial/.

[39] 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.

[40] 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.

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

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

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

[44] Viviano, A. and Bazan, N. *User mode and kernel mode - Windows drivers | Microsoft Docs* [online]. 2017 [cit. 2020-04-22]. Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode.

[45] 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.

[46] 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.

Appendix A

Contents of the included storage media