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

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

Bachelor's Thesis Specification ||||||||||||||||||||||
23591

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 vzdialeným 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 .

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

April 26, 2021

Acknowledgements

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 login, prove their identity to the *server*, and the application on the user's device, the *client*, takes care of the rest. After successful authentication, the user's files are available to read, download, and upload. Manually, these actions can cause a lot of overhead as the file count increases. What if the number of users that have access increases as well?

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

WinFsp[28] allows for implementing a file system in the userspace for Windows and offers both lower and higher-level APIs to work with. As such, the implementation of the VDU client-side application (VDU Client) is programmed with C++. **[[CONTINUE later]]**

1.1 Structure

[[Add structure]]

Chapter 2

Specification

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 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

2.1 Requirements

This section summarizes the requirements of the internal VDU API documentation. The internal VDU API documentation was provided as a document

2.1.1 Endpoints

This subsection lists all available VDU API endpoints and provides an overview of their usages. Some endpoints require an *X-Api-Key* (API key) 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 an 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 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 for 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 VDU system

2.1.2 Access

Both 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 protocol to access the API. The client-side certificate is optional and allows the user to omit the client secret from the authentication endpoint.

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

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

2.1.3 Version control

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

---

[1]Transport Layer Security

Chapter 3

Theory

This chapter serves as an introduction to the required technologies for this thesis, according to the specifications, while overviewing them in an understandable manner. It intents 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.

3.1 Development for Microsoft Windows

Microsoft Windows is the most used desktop operating system on the market, consistently having more than a 70% market share among operating systems across many years, according to [2]. Being in development for many years and thanks to its great backwards 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++ are the most effective programming languages, 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 6.

3.1.1 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.

Operating System Version

The operating system version, as described by [16], can be referred to as the build number of the Windows operating system and is the application's target version. It does not always match the operating system's name. Deciding which version to target is important because of the application's backward compatibility between operating systems. The operating system version directly corresponds to the required Windows SDK version for developing applications, i.e., for *Windows 10 Professional*, the latest SDK is the Windows 10 SDK. An SDK can be compatibile with an older version of the operating system. A good example of this is the Windows 10 SDK, which supports *Windows 7 Service Pack 1* as specified by [24]. Table 3.1 lists operating system versions for popular Windows desktop operating systems.

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

Table 3.1: Example table of versions of Windows desktop operating systems

Microsoft Visual Studio

The Visual Studio Integrated Development Environment (IDE), as described by [23], is a program developed by Microsoft and is ideal for Windows desktop application development. It includes a code editor with well-written IntelliSense, debugging tools, theme 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 [22], it is free to use under Individual Licence, allowing any individual or student to work and develop their own applications. In Visual Studio, projects which work together are grouped under a *Solution*. Each project in a solution can be built for different operating systems, with different build tools, and with different project properties.

Microsoft Visual C++

The Microsoft Visual C++ Toolset (MSVC), also known as the build tools, are included in Visual Studio and contain the MSVC compiler, linker, standard libraries, and headers for Windows API development as stated by [15]. It is usually best practice to develop under the latest version of build tools, which, at the time of writing, are the *Build Tools v142*.

Processor modes

According to [36], in the Windows operating system the processor can run code in two modes:

- *Kernel-mode*

- *User-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 of those components in kernel-mode share a single address space and have priviledged access to the entire system and can 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.

3.1.2 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

By standard, as specified in [25], a *Windows Word* is a 16-bit unsigned short integer, its data type is WORD and, for historical reasons, will always be guaranteed to be 16 bits long. A Double-Word is twice as long, 32-bit unsigned integer, DWORD. To support the new 64-bit architecture, a Quad-Word, QWORD is available. Additionally, Windows re-defines standard integers as their capitalized versions, such as INT, 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 3.1, using the standard star symbol is still a valid way to signify a pointer type while programming Windows applications, as mentioned in [25].

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

Listing 3.1: An example of declaring a pointer to a double-word

Code conventions

Windows uses *Hungarian Notation*[1], which adds semantic information to variable names in the form of prefixes.[25] The information is supposed to let the programmer know the

---

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

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 3.2.

```
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 3.2: An example of hungarian notation

Similarly, many functions expect a range of values, referred to as inputs, in their calling parameters. These inputs' semantics are not always recognizable just by looking at the variable's data type. It is often generic, meaning it holds little to no information about what exactly does the function expect its input to be. Listing 3.3 shows an example of an unclear expected input value nIndex.

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

Listing 3.3: The prototype of the GetSystemMetrics function. Source: citeWinGetSM

Character set

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

- ANSI[2] version, signified with the suffix *A*, i.e., InternetOpenA

- Unicode[3] version, signified with the suffix *W*, i.e., InternetOpenW

- An adaptive, generic version, with no suffix, i.e., InternetOpen. It is not implemented per se, rather defined as a macro, referring either to the ANSI or Unicode version, depending on the current character set.

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

Strings

The usage of strings ties closely to the current project's character set, which is either defined by a macro or set up in the project settings in Visual Studio. To take advantage of the Unicode character set when possible and fall back to ANSI, when it is not, it is a good practice to know about and use *portable run-time* functions and prototypes, according to [43]. Both prototypes and functions provide the programmer with a way to work with strings and adapt to the preferred character set automatically, recognizable by the T, _T, or _tcs prefixes. The _tcs family of functions substitutes one-to-one with wcs and str family of functions. i.e., using _tcslen substitutes wcslen for Unicode character set and strlen for ANSI character set. Listing 3.4 shows an example of all three types of definitions of a string.

---

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

```
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 3.4: An example of defining static strings

Windows

A window in terms of Windows API described by [45], is a programming construct which:

- Occupies a certain portion of the screen

- May or may not be visible at a given moment

- Knows how to draw itself

- Responds to events from the user or the operating system

By this definition, a *window* in Windows programming might not always refer to the *application window*. A button, text field, check box, or even a combo box is a window in itself. The difference is that the application window, also referred to as the *main window*, is not part of any other window of the application. The main window also often has a title bar, a minimize button, a maximize button, and other standard 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.*[45]
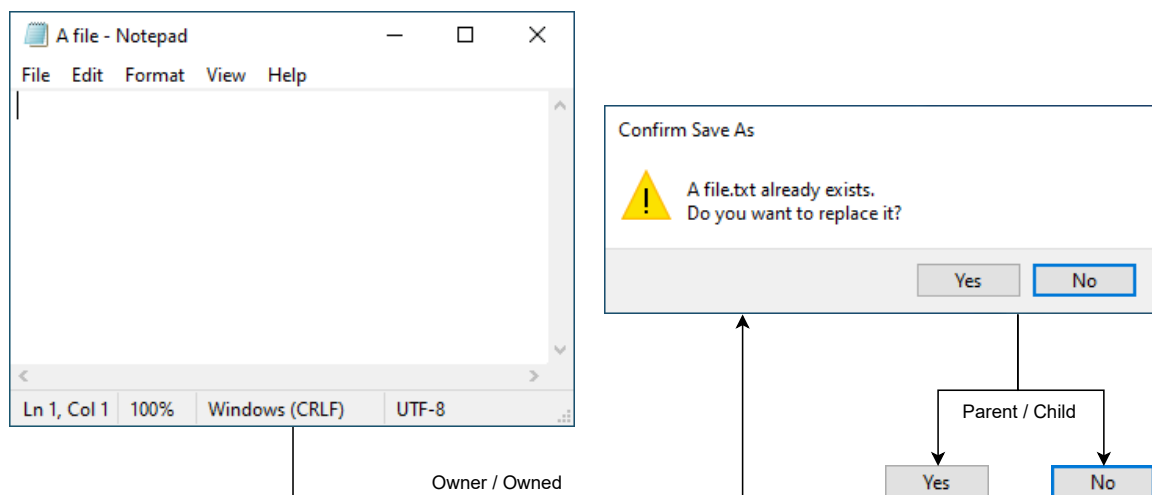

Figure 3.1: Example of owner/owned and parent/child relationships between windows.

Object handles

In Windows, there is no direct access to system resources like files, threads, windows, or graphic images like icons. These system resources are called objects and are unrelated to

9

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 [10], 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 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.[31]

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 [38], 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 [37], 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 3.2 displays the Windows registry using the Registry Editor, where some of the predefined keys, important for Windows applications, are visible:

- 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 [41]. As this project 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 [39] 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 [40], an SRW lock takes the thread's intent with the shared resource into account and is optimized for speed and performance. If a thread wants to read a resource, it can lock the resource in a *shared mode*. If a thread wants to write to a resource, it can
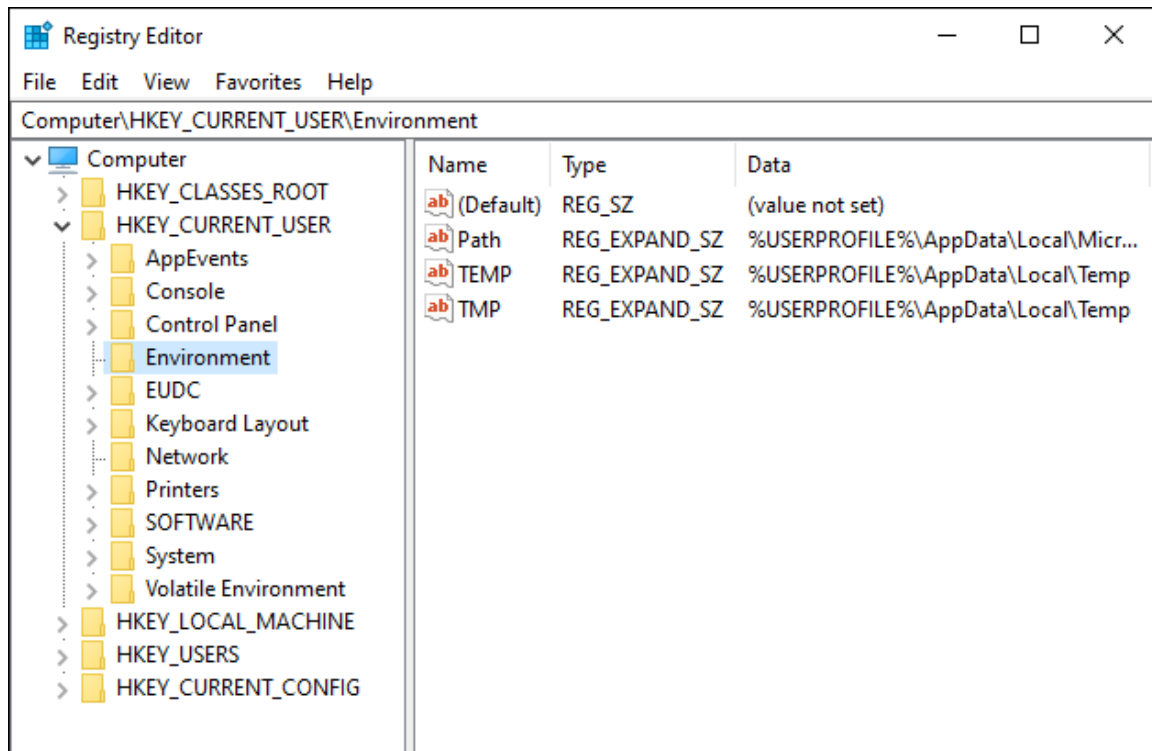
Figure 3.2: Browsing Windows registry using the Registry Editor.

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]*.*

3.1.3 Microsoft Foundation Class Library

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

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

---

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

Relations to Windows API

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

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

Listing 3.5: Showing a window using Windows API and MFC

Listing 3.5 showcases an example of showing the main window (3.1.2) using both APIs and an instance of abstracting the window handle away in favor of using a window C++ object. Calling the ShowWindow[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.

Coding conventions

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

Wrappers

Strings

Exception handling

[[Overview key MFC parts for this project]]

3.2   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.

3.2.1   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.

---

[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

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].[5]

Files in Windows are never accessed directly. Instead, applications on Windows can access a file through its handle (Section 3.1.2). When a file is opened, a handle is associated with it until the requesting process terminates or the handle is closed. Each handle is unique to each process that opens a file, and depending on which type of access to the file 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.[4]

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 Windows supported file systems have the following storage components:[12]

- *Volumes*

- *Directories*

- *Files*

A Volume is where the file system resides, is the highest level of organization in a file system, and has at least one partition, a physical disk's logical division.[44] For this project's 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.[26] It is important to remember that a directory can be referred to as a file with a special flag `FILE_BACKUP_SEMANTICS` inside the Windows API.

A file (3.2.1) 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 [34], 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

---

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

13

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 According to [6], 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 user space (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.

3.2.2 Virtual file system

A virtual file system is an abstraction of a regular file system, and as noted by [28], 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 management. To create a virtual file system without the need to create a kernel-mode driver, it is much preferrable to use a third-party virtual file system software.

Virtual file system software

The virtual file system software allows a developer of an 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 an 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 [30], there two ways in which a modern virtual file system software can provide a virtual file system API:

- A Native API

- A FUSE 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 [30].

- *Pros*: Cross-platform compatibility; easier development with a higher-level API; well-documented API

---

[9]New Technology File System

- *Cons*: Lack of Windows-specific features; restricted by POSIX standards

For this thesis, it is much preferable to choose an virtual file system software that includes a native API, as cross-platform compatibility is not a requirement. This would also mean being able to use 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 [7], 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 [8], it is 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

---

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

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 [21].

Windows File System Proxy

Windows File System Proxy (WinFsp) was created by *Bill Zissimopoulos*, and as described by [9], 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

- *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 3.3 and 3.3.

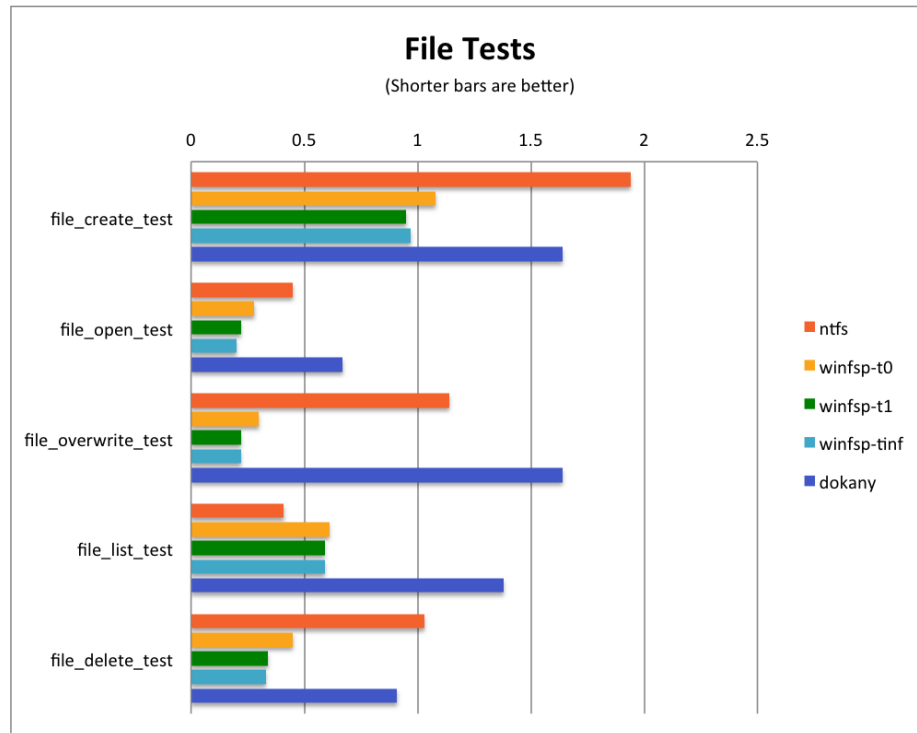[12]New Technology File System

Figure 3.3: Performance comparsion of file tests of WinFsp, Dokany and NTFS. Source:[9]
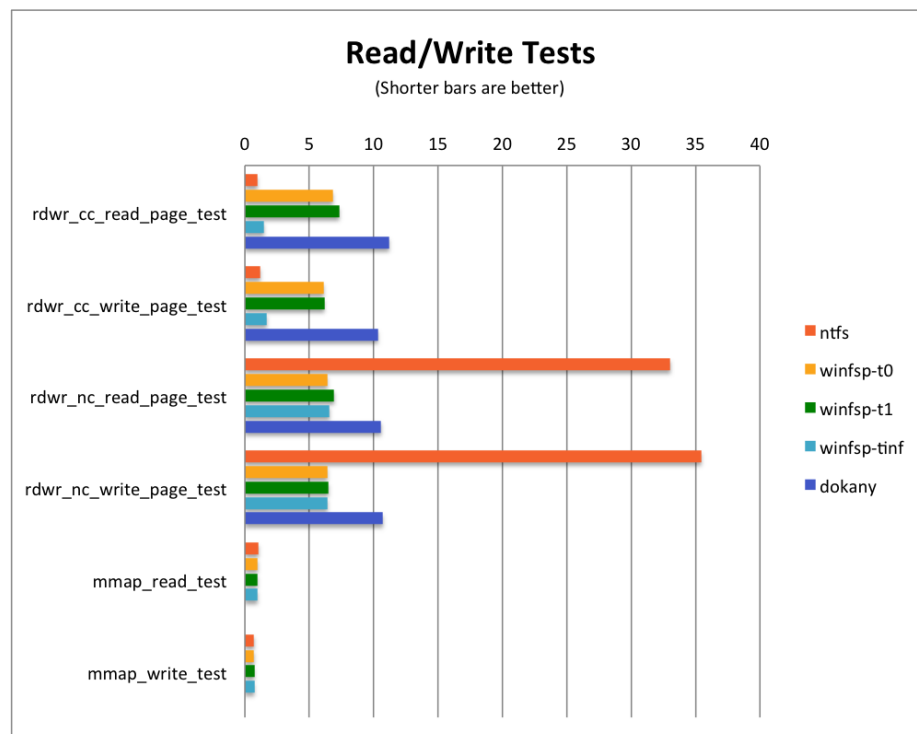

Figure 3.4: Performance comparsion of read/write tests of WinFsp, Dokany and NTFS. Source:[9]

3.3 Additional technologies

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

3.3.1 Hypertext Transfer Protocol

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

- *HTTP Request* - From client to server

- *HTTP Response* - From server to client

HTTP request

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

1. *Method* - Defines the requested operation with the resource

2. *Path* - Location of the resource

3. *Protocol version* - Version of HTTP

4. *Request headers* - Additional information about the resource

5. *Content* - Contains the content of the resource sent to the server

HTTP response

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

1. *Protocol version* - Version of HTTP

2. *Status code* - Defines the requested operation with the resource

3. *Status message* - Location of the resource

4. *Protocol version* - Version of HTTP

5. *Response headers* - Additional information about the resource

6. *Content* - Contains the content of the resource sent to the client

3.3.2   Representational State Transfer

According to [18], The Representational State Transfer represents an architectural style of developing RESTful web services, which allows the developer to take advantage of an already existing protocol. In the case of web services, this protocol is HTTP. REST conforms at the very least to the most basic REST constraints, as defined by its creator *Roy Thomas Fielding*:

- *Client-Server* - Separates the client's side and the server's side

- *Stateless* - Each request must contain all necessary information necessary to understand the request

- *Cache* - Requests must be labeled as cacheable or non-cacheable. Improves network efficiency if it is available

REST using HTTP

The key abstraction of information in REST is a *resource*. This definition is similar to the HTTP resource - a resource can be anything that can be named and might be a potential target of a request, e.g., a document, an image, a data file. The REST *endpoint* refers to the *path* of an HTTP request. The content of a resource is usually transferred as the *content* of an HTTP message. HTTP headers are useful for storing additional information about a resource, such as 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 3.2.

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 3.2: An example of a table of REST API endpoints

3.3.3   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 [20], 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

---

[13]A recursive acronym for "YAML Ain't Markup Language„
[14]JavaScript Object Notation

As Listing 3.6 shows, the OpenAPI format is easily readable and understandable for both machines and humans. Additionally, many third or first-party services provide a way to visualize the API in a graphical, user-friendly format, i.e., the Swagger Editor[15]. An example of a rendered graphical representation of the VDU server's REST API is displayed in Figure 4.1.

```
1   #A simple documentation of a /ping endpoint
2   openapi: 3.0.0
3   info:
4     version: '1.0'
5     title: An amazing API
6     description: Formal description
7   servers: #Server URL for testing
8     – url: 'https://localhost:4443'
9   paths: #Endpoint descriptions
10    /ping: #Endpoint path
11      get: #Method
12        parameters: [] #Call parameters
13        description: To test a connection.
14        responses: #Possible responses
15          '204':
16            description: Ping success!
```

Listing 3.6: An example of an OpenAPI file in the YAML format

3.3.4 Python

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

---

[15]https://editor.swagger.io/

Chapter 4

Analysis

This chapter will tackle the first step of creating an application, the analysis, and the steps taken during an analysis of the provided documentation of the VDU server. It will introduce the required technologies to understand and handle them. Results of the analysis are presented at the end of this chapter.

[[**Include google docs pdf of requirements?**]]

4.1 Formalization

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

4.1.1 OpenAPI

Formalizing the provided documentation consists of reading and understanding all API endpoints and their access or usage restrictions and manually creating an entry for each in an OpenAPI file. Each entry has its own possible status codes, headers, and content, which the endpoint could return. For this project, I used the Swagger Editor, which allowed me to document the VDU API more comfortably. The editor's great advantage is that it can render the OpenAPI specification file in an HTML[1] 5 format, as shown in Figure 4.1.

4.1.2 Conclusion

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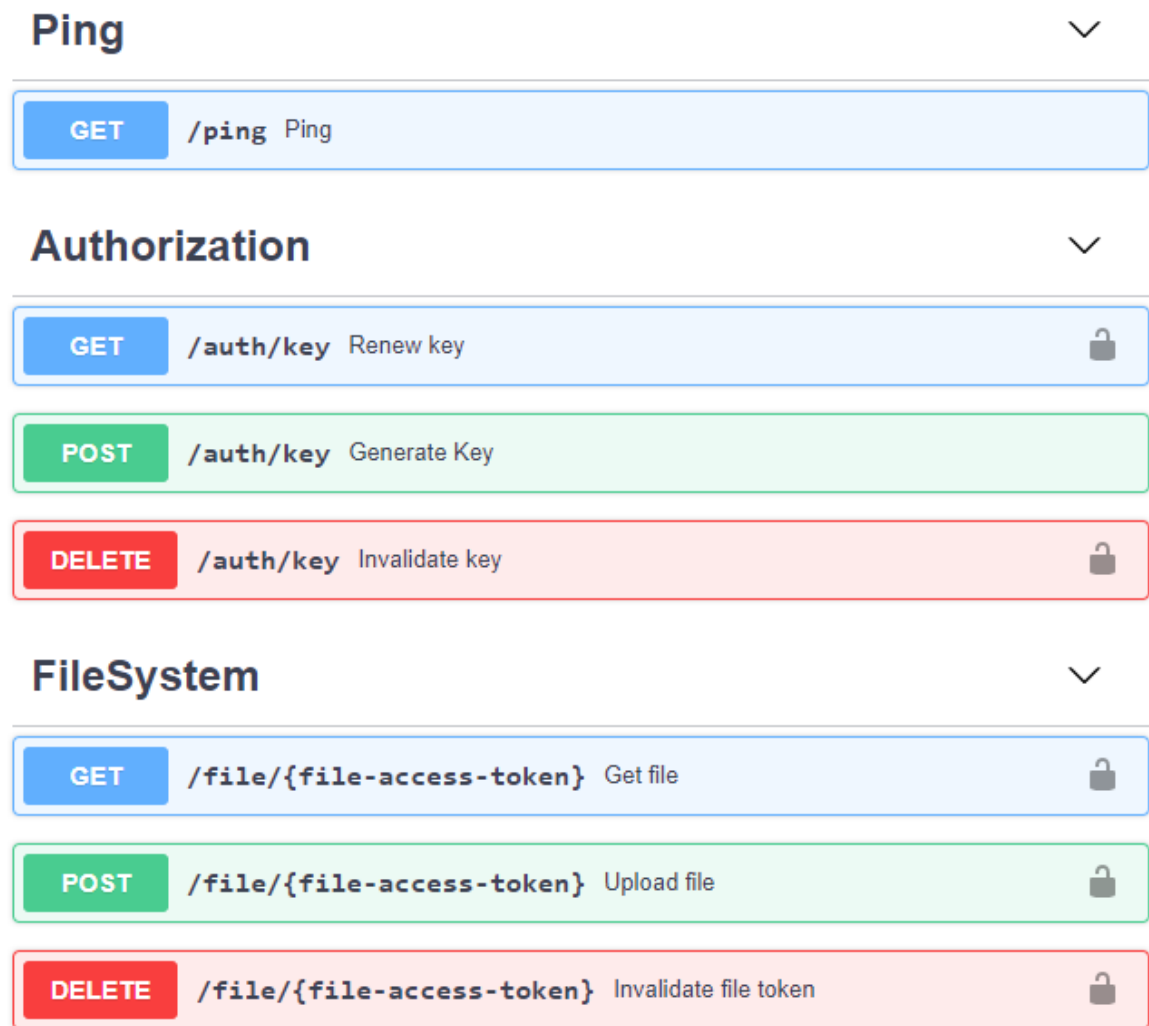
---

[1]Hyper-Text Markup Language

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

Chapter 5

Design

This chapter aims to design the application based on knowledge from previous chapters. The

5.1 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.

5.1.1 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 [1], SRP is a design principle for designing object-oriented classes or modules. The key inspieration 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 5.1 shows the class diagram of the application using the Unified Modeling Language (UML). This class diagram was created in Visual Paradigm[1].
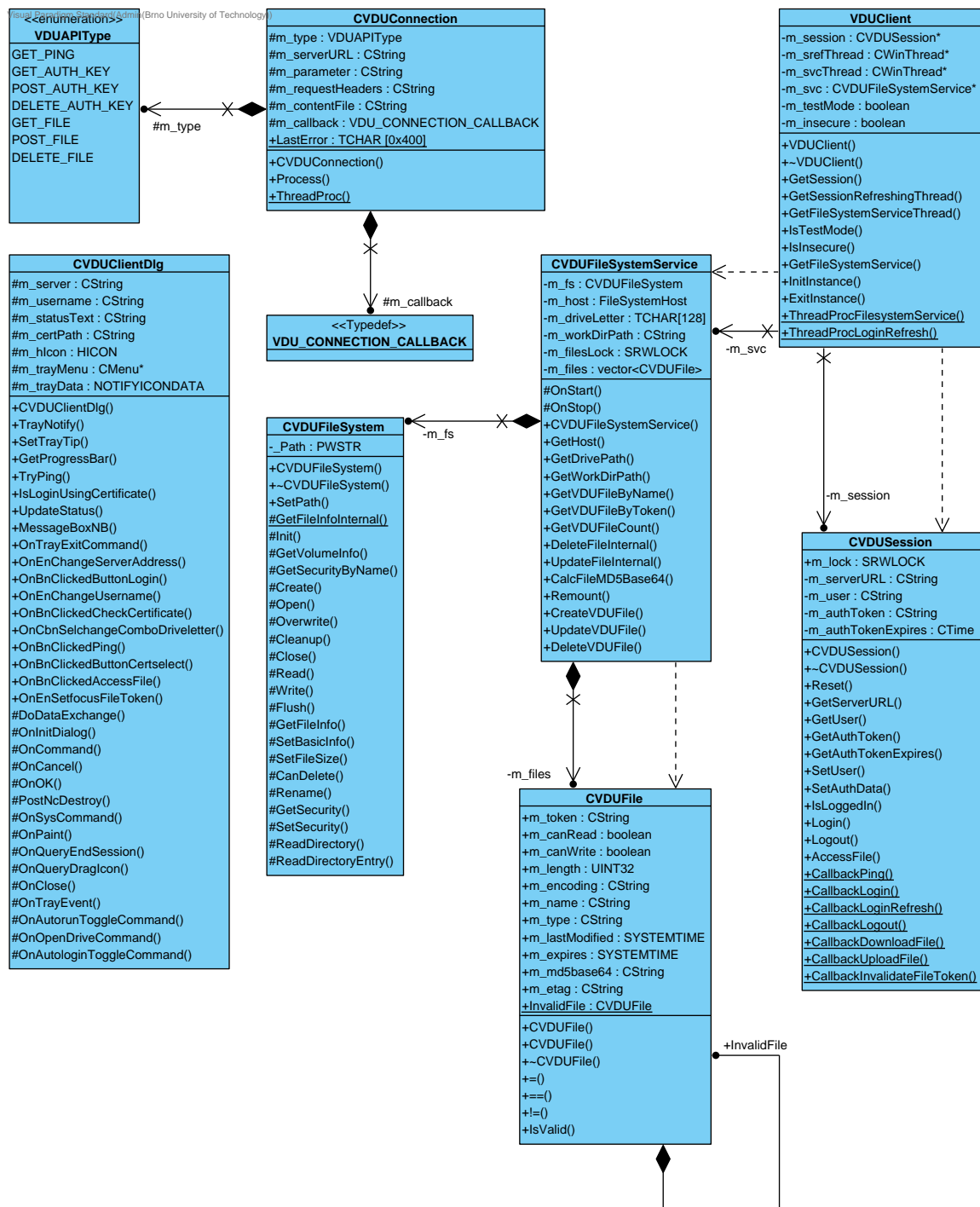
---

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

Figure 5.1: 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 can be instantiated for the VDU client to be able 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 7.

5.1.2 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, 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 fact has inspired me to attempt to store the files using just the system's Random-Access Memory (RAM) of the application. While the speed and accessibility of RAM might make this idea seem plausible, in reality, it would not work well enough for a couple of reasons:

- *Space limitations* - Large files might not have enough space

- *Inactive RAM usage* - Files not actively in use might prevent other applications from using that space

- *No file recovery* - If the system crashes, there is no way to recover unsaved work

Windows registry

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

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

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

5.2   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*:

- Test a connection to the server

- Login/out of the system

- Change the virtual drive letter

- Input a file token to access a file

- Read and modify accessed files

- Invalidate a file token of a file

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

5.2.1   Integration into Windows environment

Note that not all of these actions have to be included in the application's user interface. For example, the user can read and modify files via some other application present on the system, completely unrelated to the VDU client. This fact has led me to an important realization. Windows Explorer, a built-in tool for browsing files in Windows, has an amazing user interface that can display files provided by a file system - such as a virtual file system of the VDU client, as displayed in Figure 5.2. Using Windows File Explorer to provide the functionality of file access-related actions is an intuitive way of integrating with the Windows desktop environment.

There is one more action that Windows File Explorer is capable of - deleting files. The VDU API provides an API for file token invalidation, which has led me to make a design choice, which makes use of the delete feature of Windows Explorer, present inside the conext menu. 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 which the VDU client has to implement in its own user interface is reduced further.

In conclusion, by using built-in features of existing applications in the Windows environment, specifically Windows File Explorer, I was able to simplify the VDU client's user

interface while keeping its functionality unchanged. Details on how exactly this is implemented and why it is not limited to Windows File Explorer only are explained in chapter 6.
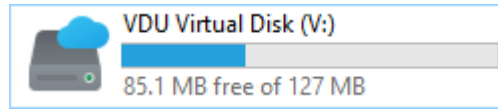

Figure 5.2: The VDU Virtual Disk as shown in the Windows Explorer.

5.2.2 Dialog design

After narrowing down all user actions, as shown in the use case diagram in Figure 5.3, the next step is to design the actual VDU client interface.
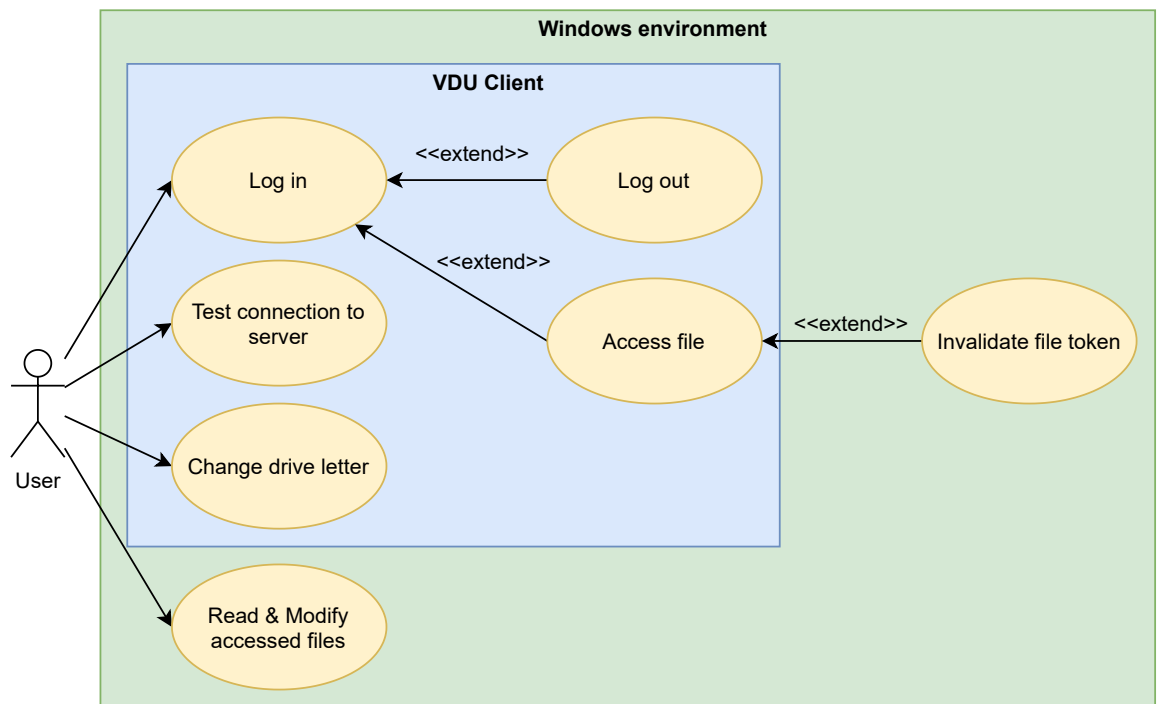

Figure 5.3: Use case diagram from the view point of an user of the VDU Client system and the Windows environment it resides in.

Considering the little functionality required, I decided to design a simple *Extended Dialog Window* (dialog). The dialog, seen in Figure 5.4, 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.
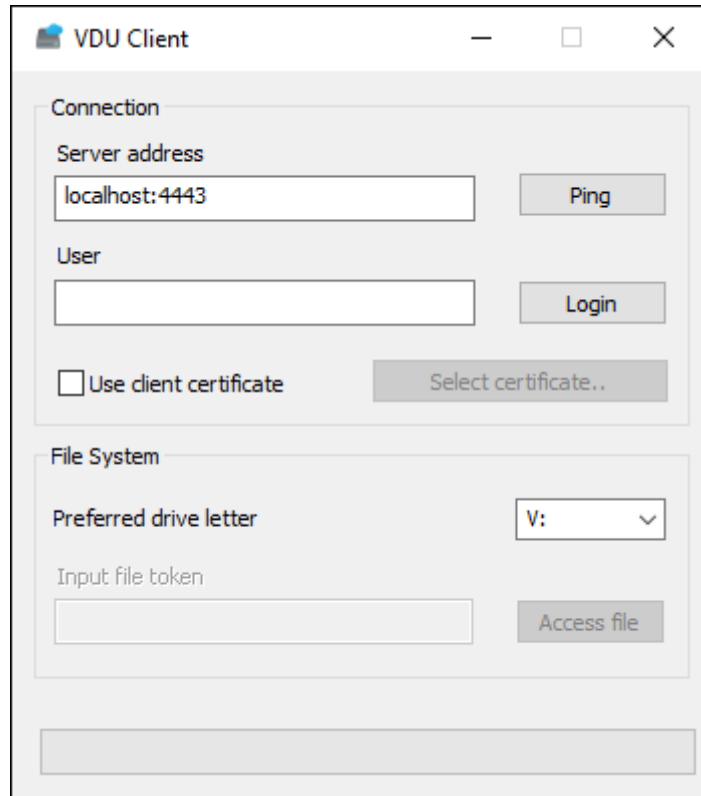
Figure 5.4: User interface of the VDU Client application.

Connection section

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

File system section

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

Status section

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

5.2.3 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, which is signified by the icon being present. The user can restore the dialog window by simply clicking on the VDU Client icon, provided by Icons8[2], displayed in Figure 5.5, in the tray area.

Figure 5.5: Cloud Storage Icon, used as the main icon for VDU client application. Source:[11].

Furthermore, removing the ability to close and exit the application from the dialog window has moved this responsibility to the tray icon. I designed a simple tray menu to fix this problem, depicted in Figure 5.6, which becomes active after right-clicking the icon. The *Exit* option in this menu, shown in the listing, is how the user is supposed to quit using the application.
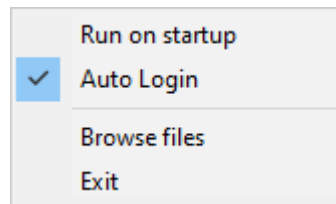

Figure 5.6: Tray menu of the VDU Client application.

Windows allows applications that put icons in the tray area to display a small text message while hovering the icon - a *tray tip*. I utilized this tray tip to display a compact text version of the data from the status section 5.2.2 of the dialog, as displayed in Figure 5.7.

Figure 5.7: Example of a tray tip, displaying the state of the application.

5.2.4 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[3]. For example, an instant response to clicking the *Ping* button, as displayed on the user interface in Figure 5.4, would disable the button, making it non-clickable. This button would be then enabled once again along with the follow-up - delayed response. The delayed response consists of either creating a message box or displaying a Windows notification.

Message box

A message box in the VDU client is a simple window, often created as an *owned window* relative to the main window. It is used as either an instant or a delayed response to important actions caused directly by the user, i.e., trying to test a connection to an incorrect server will result in a message box depicted in Figure 5.8.
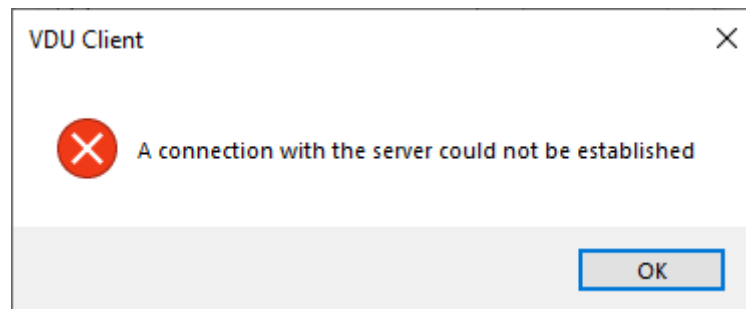

Figure 5.8: A message box window, displayed after the application fails to connect to a server.

Windows notifications

For less important and rather informative actions, a Windows notification shows up as a response. Such a notification, displayed in Figure 5.9, appears, for example, after a successful download of a newly accessed file, along with an automatic startup of the assigned application to the file type. This lets the user know it was the VDU client which caused the application to open.
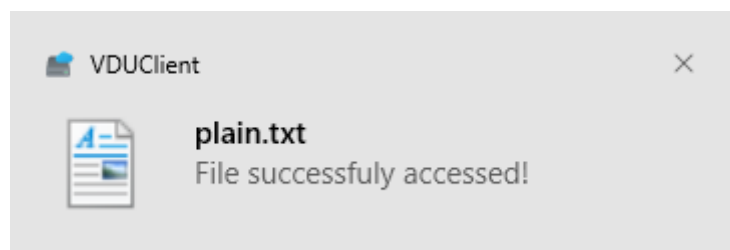
---

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

Figure 5.9: A windows notification, displayed after a successfully obtaining access to a file in the bottom right corner of the screen.

Chapter 6

Implementation

This chapter covers the exact internal implementation of key elements of the VDU client, as described in previous Chapter 5, which covered the application's overall design. The entire implementation of the application was done using Visual Studio Community 2019; an IDE[1] described in section 3.1.1. This chapter makes implicit use of the following macros:

- *WND* - Main window object (**CVDUClientDlg**)

- *APP* - The application object (**VDUClient**)

6.1 Internal components

The back-end implementation of the application's internal components corresponds to the class structure designed in Chapter 5. This section covers the details of important parts of the implementation related to the required application functionality.

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

---

[1]Integrated Development Environment

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

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

Threading connections

The application has, by default, only a single thread available, the thread which handles the user interface - the main thread. Processing connections on this thread would block the user interface from responding 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 6.2, is a non-blocking operation, ensuring that the main thread's execution flow will not be disrupted by issuing requests to the server.

```
1  LPVOID pCon = (LPVOID) new CVDUConnection(GetServerURL(),
       VDUAPIType::POST_AUTH_KEY, CVDUSession::CallbackLogin, headers, _T(""), certPath);
2
3  AfxBeginThread(CVDUConnection::ThreadProc, pCon);
```

Listing 6.2: 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 6.3. The main thread is excluded from this restriction, as it must not be blocked, and the worst-case scenario is only potentially outdated visual information.

```
1  CVDUSession* pSession = APP->GetSession();
2
3  //Blocking if already acquired, until released
4  AcquireSRWLockExclusive(&pSession->m_lock);
```

---

[2]Slim Reader/Writer lock

```
5    //Entered a critical section
6
7    //Code which uses the session data exclusively
8    CString token = pSession->GetAuthToken();
9    ...
10
11   //Leaving critical section
12   ReleaseSRWLockExclusive(&pSession->m_lock);
```

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

Callback functions

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

- *Executed asynchronously* - Executes in a worker thread

- *The parameter is the response* - The HTTP response can be NULL on failure

- *Exclusive access to session data* - To prevent data racing

- *Return value is thread exit code* - For synchronous operations

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

Listing 6.4: The prototype of a VDU callback function.

Refreshing authorization token

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

To solve this issue by creating a permanent worker thread on 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.

6.1.2 Virtual file system

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

---

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

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

- *GetVolumeInfo* - Volume information

- *GetSecurityByName* - File metadata and security descriptors

- *Create* - Creating a file

- *Open* - Opening a file

- *Overwrite* - Overwriting an existing file

- *Cleanup* - Situational file operations

- *Close* - Closing a file handle

- *Read* - Read bytes from file

- *Wite* - Write bytes to file

- *Flush* - Flush on disk

- *GetFileInfo* - Query file metadata

- *SetBasicInfo* - Set file attributes, file times

- *SetFileSize* - Change file size

- *CanDelete* - Whether or not can file be deleted

- *Rename* - Renaming a file

- *GetSecurity* - File's security descriptor

- *SetSecurity* - File's security descriptor

- *ReadDirectory* - Reading directory data

- *ReadDirectoryEntry* - Listing through directory contents

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.

Files

The VDU files are remotely accessible through the VDU server API.

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.

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 6.5 shows the implementation of the access right check.

```
1   CVDUFile vdufile = ... //Requested file
2   UINT32 GrantedAccess = ... //Requested access to the file
3
4   //If the file is a VDU file
5   if (vdufile.IsValid())
6   {
7       //If the file is read−only, check writing rights
8       if (!vdufile.m_canWrite &&
9               (GrantedAccess & GENERIC_WRITE ||
10               GrantedAccess & FILE_APPEND_DATA ||
11               GrantedAccess & FILE_WRITE_DATA))
12       {
13           //Return a descriptive NTSTATUS
14           return STATUS_MARKED_TO_DISALLOW_WRITES;
15       }
16  }
```

Listing 6.5: Implementation of the read-only check for files, to disallow creating a handle with write access.

Uploading files

Normally, local VDU files have to 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 repeatedly.

To automate this process, local VDU files present in the virtual file system will only be uploaded to the server if a change to the 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. A file can be changed in three basic ways:

- *Renaming* - Changing the file's name and or extension

- *Replacing* - Drag and drop; overwriting the file with another file

- *Direct modification* - Opened and modified by some other application

Renaming is easy to detect. If a file is about to be renamed, the 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 which handles this process. However, generally, 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. Notice that it makes use of the renaming functionality. To differentiate user-triggered renaming and renaming caused by application overhead, I use the ReplaceIfExists parameter of the Rename virtual function of the virtual file system. This parameter is False for overhead renaming and True for user-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, as described in the following section.

Deceting *direct modification* stems from the cycle of modifying files by applications. This cycle, on its most basic level, consists of the following steps, which can be intercepted in the virtual file system:

1. *Open file* - Acquire a handle to a file

2. *Modify the file* - Use the handle to modify the content

3. *Close file* - Close the 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, instead of intercepting in the middle

where the data is still being written, intercepting the end of the modification when closing the handle to a file is the best spot. The virtual function **Close** provides the handle, which is about to be closed via the **FileDesc0** parameter. To figure out whether or not this file is one of the VDU files, the **GetFinalPathNameByHandle**[4] function provides the file name of the file this handle belongs to, which can be checked against the internal vector of VDU files. However, it is important to note that this approach detects *every* handle that belongs to a VDU file. If an application 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 false-positive, the application needs to figure out whether a handle has written access. The internal Windows API function **NtQueryObject** allows querying the **GrantedAccess** of a handle object, as specified by [35]. 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 6.6. Thanks to this approach, the identifying process is more effective, as only handles with write access are considered for a file modification check.

```
1   //Prototype from documentation
2   typedef NTSTATUS (NTAPI* NtQueryObjectFn)(HANDLE Handle,
3       OBJECT_INFORMATION_CLASS ObjectInformationClass,
4       PVOID ObjectInformation, ULONG ObjectInformationLength,
5       PULONG ReturnLength);
6
7   //Get the function dynamically from ntdll
8   static NtQueryObjectFn NtQueryObject = NULL;
9   if (!NtQueryObject)
10  {
11      HMODULE ntdll = LoadLibrary(_T("ntdll.dll"));
12      if (ntdll != NULL)
13          NtQueryObject =
14              (NtQueryObjectFn) GetProcAddress(ntdll, "NtQueryObject");
15  }
16
17  //Assume the handle has writing rights
18  BOOL handleHasWriteRights = TRUE;
19
20  //Query the handle object's basic information
21  PUBLIC_OBJECT_BASIC_INFORMATION pobi;
22  if (NtQueryObject != NULL &&
23      NT_SUCCESS(NtQueryObject(FileDesc->Handle, ObjectBasicInformation,
24      &pobi, sizeof(pobi), 0)))
25  {
26      ACCESS_MASK GrantedAccess = pobi.GrantedAccess;
27
28      //Check handle's access for write flags
29      handleHasWriteRights = FALSE;
30      if (GrantedAccess & GENERIC_WRITE ||
31          GrantedAccess & FILE_APPEND_DATA ||
32          GrantedAccess & FILE_WRITE_DATA)
33      {
```

---

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

```
34          handleHasWriteRights = TRUE;
35      }
36  }
```

Listing 6.6: Implementation of accessing NtQueryObject function and using it to identify wheter the handle has access to writing.

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 as well, so 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. Thus, all it took to repurpose the delete function was to 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 real drive and a virtual one, created by the application, I implemented a simple solution, which upon mounting a virtual drive to a drive letter, sets the icon to match the VDU Client main icon, displayed in Figure 5.5, 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 [27], for Windows versions other than *Windows 2000*, which is this application's case, the key is located under the HKEY_LOCAL_MACHINE root key and requires administrator permissions to be written into.

To avoid this, the key used in the Windows 2000 option's case can be used and is working as intended, with a modification. According to [32], the HKEY_CLASSES_ROOT key can be swapped for the HKEY_CURRENT_USER\Software\Classes key, if the intent is to write

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 6.7.

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

Listing 6.7: Implementing a custom drive icon for drive Z, using without administrator permissions.

6.2 User interface

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

6.2.1 Dialog window

The dialog window was implemented using MFC[5] in Visual Studio, which allows creating dialog interfaces in a schematic-like format, as displayed in Figure 6.1. Each designed window serves as a template for programmatically created windows of the same type and has a unique *name*. When creating a window, a template can be specified to guide the design of this window. As shown in Listing 6.8, the **Create** function of **CDialogEx** is using the designed window template specified as the first parameter.

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

Listing 6.8: 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 6.9.

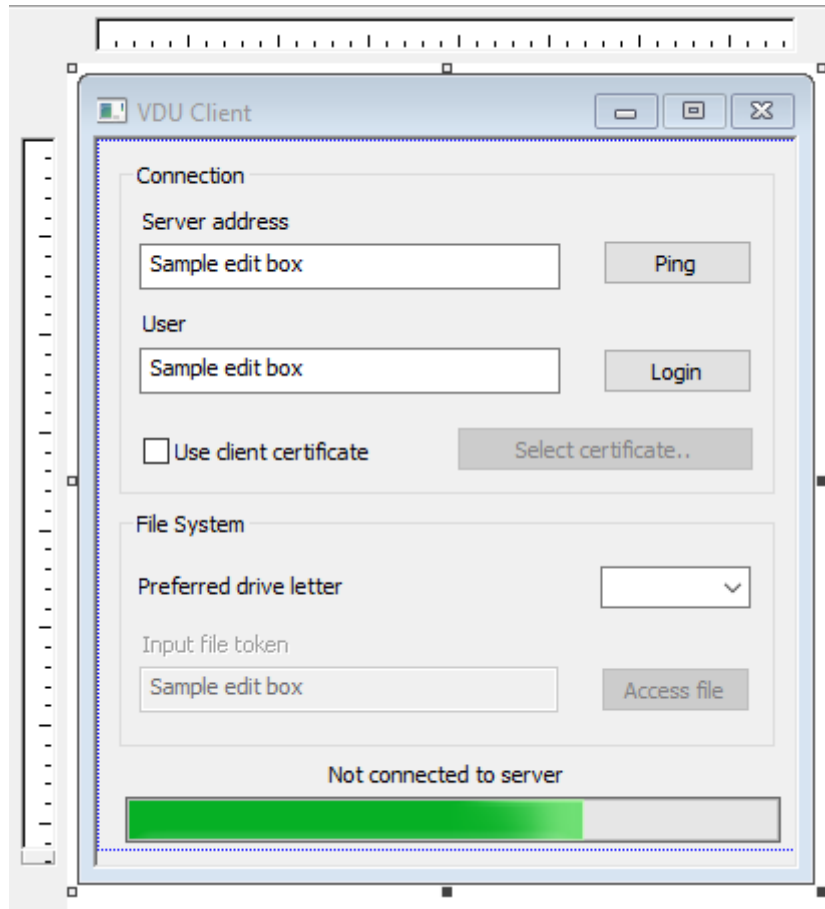[5]Microsoft Foundation Class library

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

```
1  //Message mapping for dialog
2  BEGIN_MESSAGE_MAP(CVDUClientDlg, CDialogEx)
3      ON_BN_CLICKED(IDC_BUTTON_PING, &CVDUClientDlg::OnBnClickedPing)
4  END_MESSAGE_MAP()
5
6  //The message handler function
7  void CVDUClientDlg::OnBnClickedPing()
8  {
9      TryPing();
10 }
```

Listing 6.9: Example of implemented and mapped click event handler

Message box

A message box is a small dialog which 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, or even a deadlock if a message box function fails to display the dialog and its thread does not release the synchronization lock.

To prevent this while still making use of message boxes, each message box is created in a separate new worker thread. The calling thread creates a structure consisting of the parameters of the message box and passes it to the new thread. This approach avoids any potential delays in thread synchronization.

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 to minimize and hide the main window, as shown in Listing 6.10.

```
1  //Overriding the OnSysCommand virtual function
2  void CVDUClientDlg::OnSysCommand(UINT nID, LPARAM lParam)
3  {
4      if ((nID & 0xFFF0) == SC_CLOSE)
5      {
6          //Hide to tray
7          ShowWindow(SW_MINIMIZE);
8          ShowWindow(SW_HIDE);
9          return;
10     }
11
12     //Call parent's default implementation
13     CDialogEx::OnSysCommand(nID, lParam);
14 }
```

Listing 6.10: Overriding system close command to hide the dialog window

6.2.2 Windows environment

This subsection covers features that tie closely to the interface of the Windows environment. Their presence is not explicitly required by the specification, 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 does get created locally and is available on the virtual file system's virtual drive. It might not be clear how to access this place or how to 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[7] function to open the file at its location. I also used this function to open the Windows File Explorer at the path of the

---

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

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

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 6.11.

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

Listing 6.11: Examples of using ShellExecute to open and browse files

Detecting WinFsp

The application requires WinFsp to be installed on the target system. Without its presence, the application will fail to function properly, which could lead an user who forgot to install it to conclude to report a bug that is not present.

My approach was to check for the presence of the WinFsp service in the system. According to [33], all Windows services present on the system are located under the HKEY_LOCAL_MACHINE key in the CurrentControlSet subkey. While the source and the predefined key imply a privileged or administrator access to the list of services in the registry, it is accessible to regular users as well. By observing the service list on a system with WinFsp installed, it can be noted that WinFsp has a subkey entry, which the application can attempt to read as shown in Listing 6.12, proving whether the service exists.

```
1   BOOL VDUClient::InitInstance()
2   {
3       ...
4       CRegKey key;
5       if (key.Open(HKEY_LOCAL_MACHINE,
            _T("SYSTEM\\CurrentControlSet\\Services\\WinFsp"), KEY_READ) !=
            ERROR_SUCCESS)
6       {
7           MessageBox(NULL, _T("WinFsp is not installed on the system!\r\nPlease download it
                from http://www.secfs.net/winfsp/rel/"), TITLENAME, MB_ICONERROR);
8           return FALSE;
9       }
10      key.Close();
11      ...
12  }
```

Listing 6.12: Implementation of the WinFsp service detection.

Chapter 7

Testing

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

7.1 Server simulation

Due to the network connectivity requirements of the client-server type of applications, it is difficult to provably test the functionality and properties of the client without the presence of the server. Additionally, the risk of flooding the production server due to automated testing might lower the quality of service for real users. This is the motivation behind creating a *simulated VDU server*, which would only be present locally and respond to the client's requests. This approach comes with several advantages:

- *Independence* - Even if the VDU server stops working, testing can continue

- *Interface stability* - A sudden change in the interface of the VDU server will not affect the application

- *Flexibility* - Application can be modified and properly tested over time to comply with any changes on the server

- *Transfer speed* - Transferring of files is limited by the speed of the local network

7.1.1 Specifications and analysis

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

---

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

- *Hosts an HTTP server with a secure TLS channel*

- *Can read and modify files*

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

7.1.2 Design

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

Simulating endpoints

Knowing the server's exact implementation would make the simulation more precise. It was unknown for this thesis. Thanks to the properly detailed specification and description of the VDU API, simulating every single endpoint was still possible to a reasonable extent.

7.1.3 Implementation

The server script

7.1.4 Supporting HTTPS/TLS

[[Should I include how TLS works in the theory part? Certificates, CAs, asymmetric encryption, and stuff seem a bit too much, maybe?]] Since the real server uses HTTPS protocol to communicate with the client, the simulated server should support the secure channel connection. To start the Python HTTP server in a way, which allows for HTTPS connections with the possibility of accepting client certificates, requires three important parts:

- *Certificate authority* - used to generate certificates

- *Server certificate* - contains the server's public key

- *Server key file* - contains the server's private key

It would be wasteful to acquire certificates from legitimate authorities, as the simulated VDU server will not be used for any purpose other than testing. Instead, I decided to solve this problem by creating a testing Certificate Authority (CA) using OpenSSL[4]. After acquiring a testing CA, I used OpenSSL to issue a testing certificate for the simulated VDU server. This operation generates both a public and a private key.

With CA and the server certificate generated, I wrapped the Python HTTP server's socket to use those certificates to validate connections with TLS, and to not require a client certificate from incoming connections for testing purposes. This operation allows the VDU application to successfully connect to the simulated server, assuming the application will either ignore the invalidity of the testing CA or have the testing CA installed in the system as trusted. This subsection was created and implemented with the usage of [19].

---

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

7.2 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 in an automated manner, as issuing commands to the GUI[5] is not practical. A much better solution for this problem is to create a special mode in which the application can be launched - the *test mode.*

7.2.1 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.*

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 file by token 'token'

- **–deletefile [token]** - Invalidate file by token 'token'

- **–rename [token] [name]** - Rename an accessed file, recognized by token 'token' to a new filename 'name'

- **–write [token] [text]** - Write a Unicode string 'text' at the beginning of the accessed file, recognized by token 'token' and upload it to the server

- **–read [token] [cmpText]** - Read a Unicode string of the length of string 'cmpText' from the beginning of the accessed file, recognized by token 'token', and assert that it is equal to 'cmpText'

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

Execution

The application, while in test mode, starts executing instructions from the command line. It reads all launch options of the application present in the command line, in order, one by one, until the end of the command line is reached. Every instruction is executed synchronously - it is checked whether it has been executed successfully, before continuing execution with the next instruction. After all instructions are executed successfully, the application exists with exit code **EXIT_SUCCESS**. If any instruction fails to be executed successfully, the application exits with exit code **EXIT_FAILURE**.

---

[5]Graphical User Interface

7.2.2 Testing script

To automatically test the application in the test mode, I created a script in Python which would automatically execute a number of tests. These tests assume a predefined state of the server. This state is exactly present in the simulated VDU server on its startup, making it an implicit choice for testing.

Creating a test

Using the test mode instructions, it is possible to create an unlimited number of different variations of tests for the application. These can range from unit tests to specific complex tests. Consider the following test, presented as a Windows command to run the application with launch options in Listing 7.1.

```
1  vduclient.exe –insecure –testmode –server 127.0.0.1:4443 –user John –accessfile a1b2 –write
       a1b2 book –read a1b2 book –deletefile a1b2
```

Listing 7.1: An example command with launch options to test the VDU Client application.

The step-by-step execution of this test is as follows:

1. Disable certificate verification

2. Enable test mode

3. Set server address to 127.0.0.1 with port 4443

4. Login as user John

5. Access a file recognized by token a1b2

6. Write string book at the beginning of file recognized by token a1b2; assure it gets uploaded

7. Compare whether the string of first 4 characters, at the beginning of file recognized by token a1b2 equals the string book

8. Invalidate file token a1b2

This single test asserts many different units of the required functionality of the application, including file integrity, version control, etc.

Implementation

The implementation of automated tests is present inside the testing script. Tests are started and their exit code is checked using the subprocess[6] Python library.

Running tests

---

[6] https://docs.python.org/3/library/subprocess.html

Chapter 8

Conclusion

[[**Evaluation of progress etc.**]]

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

---

[1] https://github.com/

Bibliography

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

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

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

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

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

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

[7] *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.

[8] *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.

[9] *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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] HUDEK, T., COULTER, D., KNICHEL, J. et al. *HKLM\SYSTEM\CurrentControlSet\Services Registry Tree - Windows drivers | Microsoft Docss* [online]. 2017 [cit. 2020-04-24]. Available at: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/hklm-system-currentcontrolset-services-registry-tree.

[34] 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/.

[35] *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.

[36] 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.

[37] 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.

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

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

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

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

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

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

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

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

Appendix A

Contents of the included storage media