HELSINKI METROPOLIA UNIVERSITY OF APPLIED SCIENCES

Bachelor's Degree in Information Technology

Bachelor's Thesis

Design and Implementation of a Portable Virtualization System

Author: Simon Allaeys

Instructor: Kari Björn

Supervisor: Kari Björn

Approved: __. __. 2009

Kari Björn

Principal Lecturer

PREFACE

This document is the report of my internship I completed at Helsinki Metropolia.

First of all I want to thank staff members of Metropolia, especially Mr. Kari Björn, the head of the department ICT of Metropolia, for supporting me with my thesis assignment.

I also would like to thank staff members of Katholieke Hogeschool Sint-Lieven Belgium, for giving me the chance to come to Helsinki as an exchange-student. Without their help and support, this thesis would have never been realized.

Furthermore I also want to thank Jonas Delrue and Tim Despiegelaere for allowing me to continue our work that we realized during the Projecten II course at Katholieke Hogeschool Sint-Lieven Belgium.

Christoph Husse also deserves big thanks for developing the EasyHook library as a free open source project. This library is found as indispensible for this thesis.

And last but not least, I want to thank my parents for the support they gave me during my 4 months stay in Helsinki.

Helsinki 26.05.2009

Simon Allaeys

Metropolia

ABSTRACT

Name: Simon Allaeys

Title: Design and Implementation of a Portable Virtualization System

Date: May 27, 2009

Degree Programme: Information and Communications Technology

Instructor: Kari Björn

The objective of this thesis is to design and implement a portable application virtualization system for applications designed to work on the Windows operating system. In order to achieve portability of the application, the server application is developed in a way that it's made fully portable. The guest applications are made portable by intercepting file system and registry calls with methods based on an in-depth knowledge of the Windows operating system.

This thesis contains two parts. The first part is the more theoretical one, covering the background sections of application virtualization. While the second part covers the more practical part of this thesis, it's mainly the documentation of the design and implementation of the application virtualization system as implemented in the application's source code.

There are four appendices attached to this thesis. The first one contains flow charts of the system's functionality as designed and implemented. The second appendix is a complete overview of the registry hives. The third one describes products similar to this thesis's goal. And the last one is a reference to the external libraries used in the implementation of this thesis. Also, a CD is attached to this thesis, containing the resulting source code of the implementation of this thesis.

Key words: Virtualization, Portability, Code Injection, API Hooking, C#, C++

TABLE OF CONTENTS

1 INTRODUCTION

The topic of this thesis is the design and implementation of a portable application virtualization system. The part about the design is completely documented in this document, while the implementation can be found on the attached disc. The name of this thesis's application – the implementation part - is AppStract. AppStract is a portable serving application enabling the users to apply application virtualization on virtually any of their existing applications.

Portability means that the application must not store data in the Windows registry and must not store data in any other location than the application's installation directory. Real portable applications must be developed with the goal to be portable. Making non-portable applications portable, without requiring code changes, is called Application Virtualization. Application virtualization is a term describing how applications can be isolated and made portable, independent of the hosting platform. In Windows, Application Virtualization can be achieved by transparently intercepting file system and registry calls. This term is further described in section 2.1. AppStract sets only 2 requirements. The host must support .NET 3.5 SP1 and the to-be-portable application must be compiled for the Windows platform.

An implementation for Mono, a platform independent free open source implementation of the .NET framework, is possible but will need another layer of abstraction in order to be able to translate requests for Windows resources to requests for Linux resources. Another approach could be realized by linking AppStract and Wine, but this would require an in-depth study of both technologies and a profound cooperation of both parties.

1.1 Motivation

Currently, there's a growing need for application virtualization served by applications like AppStract. The market is still quite small, but is growing rapidly while enterprises like for example VMware, Microsoft, and Symantec try to get their share in the market. In November 2008 a survey [23] revealed that three quarter of the medium-sized enterprise have adopted some form of virtualization, of which 4% have adopted application virtualization. In addition, this study showed the fastest growing area for virtualization is application virtualization, with 64 percent of medium-sized organizations reporting they have either deployed application virtualization or plan to do so in the next year. This study also indicates that many enterprises feel a growing need for application virtualization, software isolation, and software portability. Furthermore, these terms are already interpreted as synonyms for security, privacy, and maintainability by many system administrators.

Deploying a portable abstraction system like AppStract can bring many benefits in an enterprise environment. In such environment it's beneficial when users can easily use their applications on any computer, without leaking resources which might have to be kept inside of the enterprise. The use of virtual applications also reduces test cycles and deployment issues. When a virtual application works on one computer it is certain that it will work on any other computer with the same operating system version. More of the benefits brought by application virtualization are discussed in chapter 1.2

On a personal note, designing and implementing an application virtualization system might be beneficially for my market value as a person. Being able to promote myself on my Curriculum Vita with the design and implementation of a portable virtualization system during a period in which application virtualization is/was still considered as a cutting edge technology, will definitely improve my career chances in the software development and virtualization sectors.

1.2 Key Benefits of Application Virtualization

Using application virtualization in an enterprise environment brings many advantages [7][8] to both the system administrators and the end-users. Application virtualization simplifies application deployment dramatically. It eliminates application conflicts, since no application needs to be installed and each application runs in its own virtual environment. It enables system administrators to apply a dynamic license management system, which gives the users the ability to use their licensed software on virtually any computer without the need for multiple licenses. This makes applications extremely accessible for the end-user; He or she can use the available virtual applications on virtually any host and from any source. On a first glance, offering end-users the ability to use the same applications on any computer might seem to introduce some new dangers like the unwanted spreading of data which holds enterprise secrets and must be kept internal. But due to the isolation of the applications it is assured that no personal information is leaked on any host. Application virtualization might even offer enhanced privacy when compared to normal application usage since users don't have to copy their data anymore when they want to use different computers, they will just have to take their virtualized application with them and that application might even be encrypted and password protected. Another advantage is that when a user damages his version of a virtual application, the application can be instantly repaired or replaced by a clean copy. Damaged applications are inevitable in enterprise environments, and application virtualization might save a lot of time for the users/employees. Due to the enhanced security offered by application virtualization it might also avoid other applications to be damaged. All virtualized software, which might be un-

trusted, is contained in an isolated environment, protecting other applications on the system.

But there are also some disadvantages [8]. When applying application virtualization the software needs to be packaged before deployment, the system administrator isn't able anymore to just distribute the .MSI files using a system like Active Directory. Also, not all software can be virtualized, for example anti-virus applications. Another disadvantage is the extra overhead introduced by the virtualization layer, which demands an increased resource requirement compared to normal application usage.

Although, many system administrators will agree that the benefits outweigh the disadvantages. Application virtualization is only an extension to existing software management systems. Applying application virtualization on one software package does not require all software packages to be virtualized. In some cases, application virtualization might be a substitute for other kinds of virtualization. The hardware requirement needed for application virtualization will always be higher than the requirements for normal software, but on the other hand they will be lower than any other kind of virtualization in many different situations. A reduction in hardware requirements was cited as the primary benefit achieved with virtualization by 82 percent of the application virtualization users who were part of a survey inducted by KACE [23]. KACE is a leading systems management appliance company, headquartered in California.

1.3 Background

The topic of application virtualization isn't completely new to me. At KaHo Sint-Lieven students have a course called "Projecten 2" during the first semester of the 3th year (09/2008 – 01/2009). During this course we work in groups of 3 on a project of our own choice. I worked with Jonas Delrue and Tim Despiegelaere, and we chose application virtualization as the topic of our project. This topic is roughly the same as the topic chosen for this thesis. The main difference is that we didn't do an in-depth study and that our main focus lied on the code injection topic. The virtualization implementation was mainly achieved by a trial-and-error approach.

For this thesis, I'm going to complete and document an in-depth study of all sub-topics. These topics include Inter-Process Communication (IPC), code injection, Windows API Hooking, how the Windows registry works, and how calls to the file system are processed by Windows.

2 BACKGROUND SECTIONS

2.1 Application Virtualization

Many people will link the term “Virtualization” to hardware virtualization. While in fact, there are many different types of virtualization. And many of these types can be divided in other subcategories. The following chapter discusses the main types of virtualization.

2.1.1 Types of Virtualization

For the sake of this thesis only the three most common types are discussed [7]. The fourth type, Application Virtualization, is discussed in chapter 2.1.2.

2.1.1.1 Hardware Virtualization

Hardware virtualization uses a virtual machine (VM) on which an operating system can run on. This VM is essentially an entire set of simulated hardware. Because a VM implements its own hardware, it can be easily moved around to different machines and different host OS’s.

Hardware virtualization can be either implemented with as a hypervisor or as a software package, both types use a virtual machine monitor (VMM). Hypervisors are software systems that run directly on the host's hardware as a hardware control and guest operating system monitor. A guest operating system thus runs on another level above the hypervisor. Software packages have similar functionality, but with increased overhead because of the extra management layer between the guest OS and the host OS.

2.1.1.2 Paravirtualization

The idea of paravirtualization is relatively old. In 1972, IBM was the first to introduce such a facility with commercial goals. Paravirtualization allows the VMM to be simpler and more efficient, which might lead to performance closer to non-virtualized hardware. It is a virtualization technique that presents a software interface to virtual machines that is similar but not identical to that of the underlying hardware. Such a system reduces the overall performance degradation of machine-execution inside the virtual-guest. The main drawback of paravirtualization is that the guest OS must be explicitly ported to run on top of a paravirtualized VMM. In other words, the guest OS must explicitly support paravirtualization.

2.1.1.3 OS Virtualization

OS Virtualization is a less common type of virtualization. Its use lies more within Linux-based systems. OS Virtualization doesn't simulate an entire set of hardware; instead it uses the guest's kernel for all the hosts. This means that for example calls to a Windows OS (the guest) are translated to calls to a Linux OS (the host). Each host functions like it has full control of the OS. This is achieved by 'partitioning' the host OS, which also isolates each guest from the host and/or other guests. This form of virtualization introduces little or no overhead, because there is no need for an emulation layer. It also does not require hardware assistance to perform efficiently. A drawback of this technique is that it's not very flexible since it cannot host a guest operating system different from the host one, or a different guest kernel.

2.1.2 In-Depth: Application Virtualization

"Application virtualization is an umbrella term that describes software technologies that improve portability, manageability and compatibility of applications by encapsulating them from the underlying operating system on which they are executed. A fully virtualized application is not installed in the traditional sense, although it is still executed as if it is. The application is fooled at runtime into believing that it is directly interfacing with the original operating system and all the resources managed by it, when in reality it is not. Application virtualization differs from operating system virtualization in that in the latter case, the whole operating system is virtualized rather than only specific applications." *[http://en.wikipedia.org/wiki/Application_virtualization]*

Application virtualization isolates applications in self-contained packages that are able to operate under an OS and use all of its resources. Therefore, the virtualized application must support the guest Operating System (unless the virtualization server integrates some kind of OS Virtualization). Each package is basically a complete environment for the application, isolated from the guest OS. The main benefit of this approach is that the application can't break or pollute the guest OS. Because each application runs in its own virtualized environment they no longer require installation, thus providing extremely easy software distribution along with the safety of a completely isolated application. Software conflicts are reduced to a minimum because virtualized applications can run perfectly next to regularly installed applications and next to each other virtualized applications while still being able to communicate with each other.

On a side note, the term "Application Virtualization" is also used to describe the purpose of the .NET Common Language Runtime and the Java Virtual Machine [7]. This is a

totally different interpretation and should never be confused with the term “Application Virtualization” as used by this thesis.

2.1.3 Implementation in AppStract

AppStract divides the communication of a Windows application with the guest OS into two categories. The first category is the communication with the file system, and the second category is the communication with the Windows registry. To achieve application virtualization both categories must be transparently intercepted, virtualized and isolated from the host operating system. Basically this means that there must be a virtual file table and a virtual registry. Both components and their implementations are discussed in sections 3.2 and 3.3. But to ensure processes to be and stay completely virtualized a third category is introduced, namely the creation of new processes by a virtualized process. The way this category is implemented by AppStract is documented in section 3.1.

As said, the first category is the communication with the file system. This category includes requests to file and directory resources. These resources can be opened, created, and/or deleted. This first category also includes requests to libraries, to which the Windows operating system can only open a handle. In the implementation in AppStract, the functions don’t return the pointers themselves. Instead, the return data is acquired by altering the parameters used for calls to the file system and passing them to the real functions (the trampoline functions, as described in section 2.2.2). The second category is the communication with the registry. This category is also divided in two parts. Communication with the registry can either be the opening, creating, or deleting of registry keys, or the querying, setting, or deleting of registry values. All functions related to these actions are fully virtualized. This means that, unlike as with the virtualization of the file system, the trampoline functions are used as less as possible.

2.1.3.1 Simplified Overview

The following is a visual overview of the application with brief comments on how application virtualization is implemented by AppStract. The components shown in this figure will be discussed in depth in the upcoming sections.

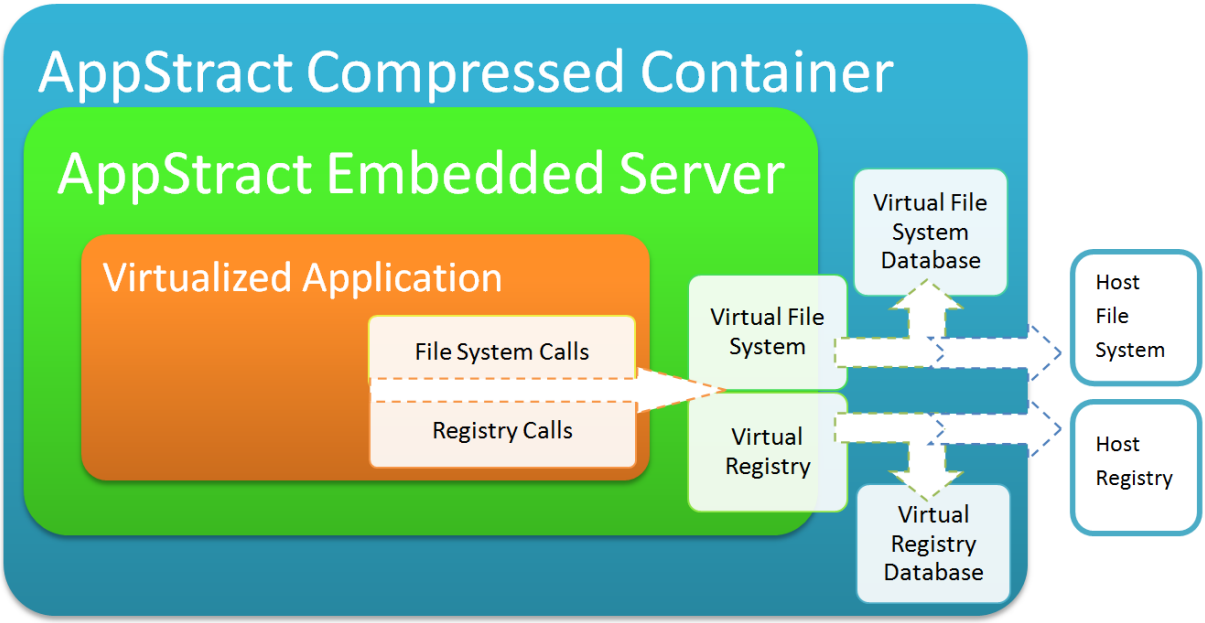


Fig. 1: Simplified Overview of the AppStract Architecture

The APPSTRACT COMPRESSED CONTAINER can either be a single file or a directory. This container contains the initialization and state data of the virtualized application. This data consists of the database files for the registry and the file system, and of the setting files needed by the AppStract Embedded Server to start a process for the application. (As documented in Chapter 3.4)

The APPSTRACT EMBEDDED SERVER exists only when the process is running. This component is installed in the process by an on-the-fly injection and is responsible for the installation of the required API hooks and their implementations (as discussed in Chapter 2.2.4).

The VIRTUALIZED APPLICATION is the virtualized process running in an isolated, virtualized environment. The application is fooled into believing that the system resources it accesses are actually real, the virtualization is fully transparent and the host application executes as if it's running directly on the host OS.

2.2 Windows API Hooking

Windows API Hooking consists of intercepting a function call in a program and redirecting it to another function. By doing this, the parameters can be modified and used with the real function, or the function can be replaced with another function without the program noticing.

2.2.1 Intercepting Function Calls

Several techniques exist for intercepting function calls [10], these include:

- CALL REPLACEMENT IN APPLICATION SOURCE CODE.
  This technique requires access to the application source code in order to replace calls to target functions with calls to detour functions.
- CALL REPLACEMENT IN APPLICATION BINARY CODE.
  The application's binaries are altered so calls to the target functions are replaced with calls to the detour functions. This requires substantial symbolic information that is not generally available for binary software.
- DLL REDIRECTION.
  The DLL import table in the application's binary can be modified to point to a detour DLL. This modification must be accomplished before load time. Unfortunately, redirecting to the detour DLL through the import table fails to intercept DLL internal calls and calls on pointers obtained from the `LoadLibrary()` and `GetProcAddress()` functions [26].
- BREAKPOINT TRAPPING.
  The target function can be intercepted by placing a debugging breakpoint into the target function. The debugging exception handler can then invoke the detour function. Intercepting via breakpoint trapping has an extremely high performance penalty, because debugging exceptions suspend all application threads while a second operating-system process catches the exception.
- RE-WRITE IN-MEMORY INSTRUCTIONS.
  In-memory code for target functions can be replaced at execution time.

The Windows operating system provides a library for API hooking that re-writes the in-memory code for target functions at execution time. This library is called *Detours* [9]. Re-writing in-memory instructions is an efficient and very flexible technique, it is documented by Microsoft that *Detours* introduces an overhead of less than 400ns on a 200MHz Pentium Pro processor [10]. When projecting the result of this benchmark to current processor architectures one might expect the overhead introduced by *Detours* to be close to zero.

2.2.2 Microsoft Detours

Detours is a library provided by the Windows operating system. The latest release of *Microsoft Detours* was in December 2006 [9]. It intended use is to intercept Win32 functions on x86, x64, and IA64 systems by re-writing the in-memory code for target functions at execution time [9]. The function call interception is temporary and exists only during runtime since the target function is modified in memory and not on disk.

Detours replaces the first few instructions of the target function with an unconditional jump to the user-provided detour function. Instructions from the target function are preserved in a trampoline function [10]. The trampoline function consists of the instructions removed from the target function and an unconditional branch to the remainder of the target function. The detour function can either replace the target function or extend its semantics by invoking the target function as a subroutine through the trampoline.

Detours is used widely within Microsoft and within the industry. As an example, Microsoft uses Detours with its *AppFix* to improve compatibility of older applications with all operating systems since Windows XP. AppFix is a system that release on the Application Compatibility Database of Windows, this is a database used to identify application compatibility issues and their solutions. With AppFix, hooks are installed for APIs called by the components of the application. These hooks point to stub functions that can be called instead of the system functions. This practice is also known as shimming. The stub functions perform operations needed to enable the application to run on the installed version of Windows. Each stub function may optionally call the system function after completing its work. A compatibility layer or mode contains one or more shims and flags. [25]

2.2.3 EasyHook

Detours doesn't provide an API for managed code, and the workload of writing such an API would be a thesis project on itself. This is where *EasyHook* comes in. EasyHook describes itself as follows:

> *"EasyHook starts where Microsoft Detours ends.*
>
> *This project supports extending (hooking) unmanaged code (APIs) with pure managed ones, from within a fully managed environment like C# using Windows 2000 SP4 and later, including Windows XP x64, Windows Vista x64 and Windows Server 2008 x64. Also 32- and 64-bit kernel mode hooking is supported as well as an unmanaged user-mode API which allows you to hook targets without requiring a .NET Framework on the customers PC. An experimental stealth injection hides hooking from most of the current AV software."* *[http://www.codeplex.com/easyhook]*

The EasyHook library brings several advantages. First of all, it's assured by EasyHook that no resource or memory leaks are left in the target while all hooks are installed and automatically removed in a stable manner. EasyHook also enables developers to write pure managed hook handlers for unmanaged API's. This brings many more benefits, it's now possible to use all the conveniences provided by managed code. Think for example about .NET Remoting, WCF, and WPF. Another major benefit of EasyHook is that the library supports the injection of libraries which are compiled for `AnyCPU`, allowing developers to use the same assemblies to inject code into both 32- and 64-bit processes from both 32- and 64-bit processes. Summarized, EasyHook greatly eases the burden of code injection by offering an extremely flexible and stable system.

EasyHook reached a stable state on the 8th of March 2009, but still brings some minor drawbacks in the managed API which should be solved in the next version (planned to be released in the 4th quarter of 2009). The major drawbacks affecting this thesis are documented in the following section and its subsections.

2.2.4 Code Injection

The run-time layer of the server application is embedded into processes by an on-the-fly library injection. This on-the-fly injection is performed before any of the process's code is executed. The injected code hooks file- and registry-related system calls. Intercepted calls are then redirected or replaced by self-implemented methods. During development I ran into some pitfalls related to code injection with EasyHook, the most important pitfalls are discussed in the subsections of this chapter.

2.2.4.1 Injecting Managed Targets

Virtualized applications mus be hooked before any of the code is executed, this is possible by using the `CreateAndInject()` method of EasyHook.

2.2.4.1.1 CreateAndInject()

The following is a brief overview of how EasyHook's `CreateAndInject()` is implemented:

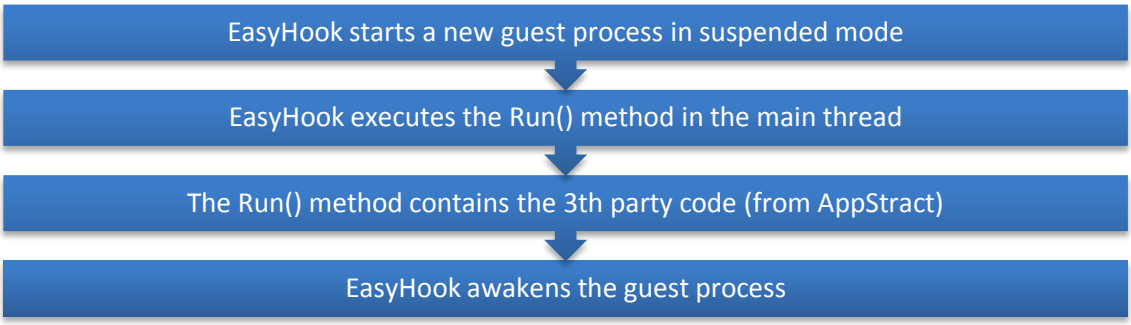


Fig. 2: `CreateAndInject()` Workflow Overview

This method works fine when it comes to injecting unmanaged targets during creation of the process, but using the same method to inject managed targets causes the injecting thread to stall.

2.2.4.1.2 Definition of the problem

The cause of this issue lies within the way the .NET framework starts processes. During the process's startup, .NET hijacks the main thread of the process to run the `Main()` method of the application. In other words, the code of EasyHook is never executed because it is supposed to run in the first/main thread which is always hijacked by .NET.

In the meanwhile the server keeps waiting for the constructor of the library's entry point to return, which will never happen, making the injecting thread stall.

2.2.4.1.3 Solution

Fortunately there is a workaround for this problem. It is possible to start a wrapper process using managed code, responsible for installing the required API hooks. These can be installed either locally or remotly. Remotly means that a library is injected in the wrapper process, and locally means that the wrapper process installs it's own hooks. Injecting a dll in the wrapper process is more flexible and simplifies the setup of the IPC because the inter-process connection is then completely instantiated and handled by EasyHook. This injected library is the same as the library used when executing an injection with `CreateAndInject()`. Using the same library makes sure the same code is used, which makes the application easier to maintain when bugs are found or addition have to be made to the injected library.

Once the hooks are installed, .NET Reflection can be used to launch the to-be-virtualized application in the wrapper process. This is achieved by loading the assembly into an application domain of the wrapper process, where then the entry point of the executable is called using dynamic invocation.

2.2.4.2 Injecting Managed Code

The complete codebase of AppStract is written in C#, which is managed code. This code-base also includes the libraries used to inject and install the hooks with.

2.2.4.2.1 Definition of the problem

The injection of libraries developed in managed code requires the assembly and all of its references to be registered with the Global Assembly Cache, which is also known as GAC. The Global Assembly Cache is used by Windows to store assemblies which are specifically designated to be shared by several applications on the computer. In other words, the GAC contains assemblies that can be shared between processes that are using different working directories. The requirement of having to register the libraries introduces two issues.

The first issue is that all assemblies must be strong named when they need to be registered with the GAC. Another term for "strong name" is "strong key". A strong name is used by the .NET framework to uniquely identify a component [6]. When an assembly is

strong-named it can only reference other strong-named assemblies. Since strong names are only used within the .NET framework, this requirement doesn't matter when referencing assemblies developed in unmanaged code. Many developers have the bad habit of never strong-naming their assemblies unless they really have to. Luckily all referenced assemblies are part of AppStract or have been developed in unmanaged code. So this first issue can be easily fixed by using only strong-named assemblies.

The second issue is that the Global Assembly Cache resides in a subdirectory of the `systemroot` directory, which is protected by an Access Control List (ACL). By default, this ACL only allows administrators to access the directory. Also, it is recommended by Microsoft that only users with Administrator privileges should be allowed to delete files from the Global Assembly Cache. Allowing users to run with Administrator privileges introduces major security issues since every user is a potential attacker and/or a potential means of entrance for remote attackers. Altering only the Access Control List protecting the Global Assembly Cache also introduces major security issues, since that would allow users to perform permanent DLL redirection on the host by replacing registered libraries with libraries containing dangerous code. Due to the security issues involved with allowing users to run with Administrator privileges, this should never be allowed by any enterprise-level system administrators. Because AppStract becomes unusable without administrator privileges a solution must be found for this problem.

2.2.4.2.2 Solution

There are two possible solutions to fix the issue introduced by having to register libraries with the GAC. We can either inject only unmanaged code instead of the existing managed code, or we can wait for a future version of EasyHook. Injecting only unmanaged code does also mean that all assemblies referenced by the injected library must also be developed in unmanaged code, including the existing core and server libraries of AppStract. Developing all libraries in unmanaged code would greatly increase the complexity of the application. *This is a pure personal opinion, since my knowledge of programming languages is mainly focused and limited on the managed programming language C#.*

In my opinion, waiting for the next version of EasyHook is the better solution. It is said by the author of EasyHook that it will be tried to let the next version utilize the CLR Hosting API, eliminating GAC usage. The release of this version is planned for the end of 2009 or the beginning of 2010. Until then the requirement to run AppStract with administrator-rights or to change the `systemroot`'s ACL will stay.

2.2.4.3 Exception handling in suspended process

Creating and injecting a process with `CreateAndInject()` causes the process to sleep until `RemoteHooking.WakeUpProcess()` is called [24]. This enables the application to make sure that the injected code had the time to install and validate the API hooks before the execution of the code of the guest process is started.

But putting a process to sleep also introduces an important and undocumented issue. Namely; Unhandled Exceptions don't cause the process to crash, instead they cause the process to stall. And due to the design of AppStract the server's creation thread is also stalled when the guest process stalls, while there's no possibility for the server to detect the stall. In the current implementation of AppStract, the user can only terminate the stalling thread by killing both the host and guest process of AppStract. In future versions a GUI element can be introduced, allowing the user to abort the stalled thread if the startup procedure of the guest process takes too long. There is no way to automate this procedure since the server's creation thread is not expected to return before the guest process exits.

The only error proof solution is to place a try-catch structure around all code preceding the `WakeUpProcess()` call, the catch clause must then log any caught exceptions and immediately kill the sleeping process.

2.3 Inter-Process Communication

Inter-Process Communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or multiple computers connected by a network. There are a number of APIs which may be used to achieve IPC in a Windows environment. For this project, I'll be using .NET Remoting [1]. .NET Remoting is an API of the .NET framework provided by Microsoft, it's released in 2002 together with the 1.0 version of the .NET Framework.

2.3.1 The need for Inter-Process Communication

Inter-process communication is needed when multiple processes have to communicate with each other on the same machine or across networking boundaries. In order to improve the security of the Windows operating system, each running process is forced to run in its own reserved memory space. When a process attempts to execute read or write actions on memory addresses outside of its reserved address range, the Windows operating system returns `exitcode 0xC0000005` which equals to an `AccessViolationException` in the .NET framework. Not being able to share data between different processes using the host's memory resources, limits the possibilities for inter-process communication significantly. An alternative could be to set up a networking connection between multiple processes and to send messages across this connection. But this method has a huge lack of performance when compared to the performance achieved by sending messages across the host's memory. Luckily Windows does provide an API enabling processes to share memory and to set up a fast inter-process communication connection. This API is wrapped and interfaced in .NET by an API called .NET Remoting.

In .NET the domain in which an isolated software application is executed can either be a process, or even just a small part of a process. Such part is referred to as an application domain in the .NET terminology. Each process may contain multiple application domains, which also may consist of multiple contexts [5]. A context guarantees that a common set of constraints and usage semantics govern all access to the objects within it. All applications have a default context in which objects are constructed, unless otherwise instructed. A context, like an application domain, forms a .NET Remoting boundary. Access requests must be marshaled across contexts using an inter-process communication connection set up by for example the .NET Remoting API.

The .NET Framework is able to optimize the use of objects when it's guaranteed that the objects will only be used inside one context [1]. When it's not guaranteed that all set of

constraints and usage semantics of an object is equal to the rest of the application domain's, optimization fails. This is the case when an object-type is derived from `MarshalByRefObject`, or is marked with the `SerializableAttribute` and/or implements the `ISerializable` interface, because such types might be used to marshal calls across remoting boundaries.

2.3.2 Inter-Process Communication in .NET

.NET offers an API for inter-process communication, called .NET Remoting. This chapter introduces the key terms and key functionalities of this API. The first item to be discussed is the way objects are made usable to be marshaled across remoting boundaries by the .NET framework. In order for an object-type (or class) to be marshaled across remoting boundaries the type must be derived from `MarshalByRefObject` , or it must implement `ISerializable`, or it must be marked with a `SerializableAttribute`. Another term used to describe remoting boundaries is application domain boundary.

2.3.2.1 Architecture of .NET Remoting

The following image is a visual overview of how .NET Remoting functions internally [5], and how remotable objects are marshaled across remoting boundaries.

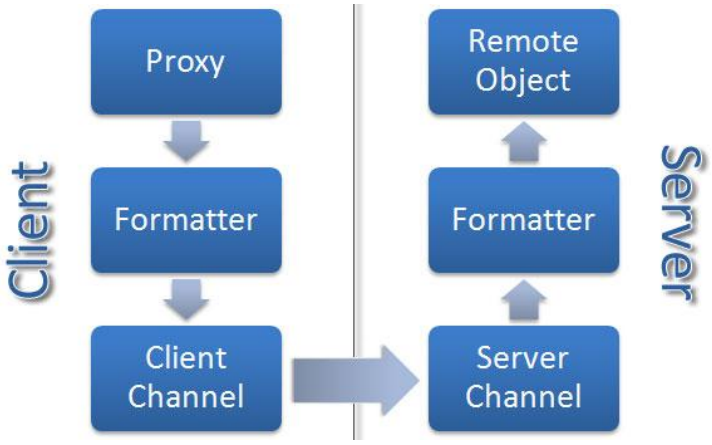


Fig. 3: .NET Marshalling Architecture, based on [5]

.Net Remoting uses PROXIES when marshalling objects by reference across Remoting boundaries. A proxy is created for an object the first time the object is referenced. The created proxy exposes the same interface as the remote object, without actually implementing any of its functionality. Instead it passes the call and its parameters as a message to the formatter.

The FORMATTER's responsibility lies in formatting method calls and objects so they are ready to be transferred across the remoting channel. Objects that need to be transferred across Remoting boundaries must be serializable in order to be compatible with the formatters. Once the message is formatted, the formatter sends it to the client channel which communicates with the server part of the channel to transfer the message across the remoting boundary. When the server receives the message on its channel, it passes the message to the formatter which deserializes the message end sends it to the REMOTE OBJECT.

Server applications that host remote objects have to register the CHANNELS as well as the objects they wish to expose with the remoting framework. This means that a remote object does not own a channel, but that they in fact share channels. The system of having to register channels and objects gives remote `SingleCall` objects the ability to listen for clients trying to connect, even if the remote object doesn't exist yet. `SingleCall` is one of the three available types for .NET remotable objects.

2.3.2.2 Types of .NET Remotable Objects

A remotable object can either be a server-activated object or a client-activated object. These two types are specified by the .NET Remoting API [3].

Server-activated objects (SAO) are objects whose lifetimes are directly controlled by the server. Server Activated Objects are not created when the client calls new, the creation only occurs when the client makes its first method call on the object. There are two activation modes for Server Activated Objects, namely `SingleCall` and `Singleton`. `SingleCall` objects are created each time the service is called. They cannot hold state information between method calls. While `Singleton` objects are objects that service multiple clients, and are able to store state information between client invocations. They are useful in cases in which data needs to be shared explicitly between clients, and also in which the overhead of creating and maintaining objects per-call is substantial.

Client-activated objects (CAO) are server-side objects that are activated upon request from the client. The server returns an `System.Runtime.Remoting.ObjRef` back to the client application that invoked it. A proxy is then created on the client side using the `ObjRef`. The client's method calls will be executed on the proxy. Client-activated objects can store state information between method calls for its specific client, and not across different client objects. Each invocation of "new" returns a new proxy to an independent instance of the server type.

2.3.2.3 Marshaling

Marshaling is the process of packaging and unpackaging object access requests that need to be passed between different application domains. The .NET Remoting infrastructure manages the entire marshalling process [4]. The API makes 2 methods available for marshalling object access requests across remoting boundaries.

Objects that are marshaled by value are created on the remote server, serialized into a stream, and recreated at the client's side. Once copied to the caller's application domain by the marshaling process, all method calls and property accesses are executed entirely within that domain. Marshal by value has several implications. The first implication is the associated reduction of performance due to the transmission of entire objects over remoting boundaries. This is especially the case when the physical form of the remoting boundary is a network connection. Another implication is that the object has no relevance outside of its local context, meaning that certain state parameters might be worthless on client-side. An example of such state information is a connection to a database. Marshalling objects by value might however increase performance and reduce network traffic if the objects are small and frequently accessed by the client. In order for classes to be able to be marshaled by value they require to be serializable.

Marshaling by reference is analogous to having a pointer to an object. When marshaling by reference a reference to the remote object is passed back to the client. This reference is an instance of the `ObjRef` class that contains all the information required to generate the proxy object that interfaces the communication with the actual remote object. On the network, only parameters and return values are passed. Marshal by reference classes must inherit from `System.MarshalByRefObject`.

When marshaling by reference, the server application domain is unable to know if an object is still referenced and used by the client application domain. This makes it impossible for the Garbage Collector (GC) to know if the memory reserved by the marshaled object may be released. The Garbage Collector is a system used in the .NET framework to manage the allocation and release of memory [5]. The garbage collector performs collections, based upon the allocations being made. The timing of such collection is hardly controllable by applications, and it should never be tried to control that timing because this might have a significant impact on performance of the whole application domain. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

The management of the lifecycle of remote objects in .NET is made possible by the existence of a system enabling parties to put a lease on remote objects. Each object has a lease that prevents the local garbage collector from destroying it, and most distributed applications rely upon leasing. The default lease time for an object is five seconds [3]. When the lease expires, the .NET Remoting Framework consults a third party - called the sponsor - giving that party an opportunity to renew the lease. To avoid expired leases, the AppStract server application keeps references to all shared objects that are marshaled by reference, until the client exits. Those types of objects are in fact referenced by the client during its whole runtime. Holding these references simplifies the use of .NET Remoting in AppStract, and also improves performance since no third parties must be consulted every five seconds.

3 IMPLEMENTATION

This chapter will discuss how the virtualization components are implemented in AppStract.

3.1 Virtualized Process

A virtualized process is a process that runs in a virtual environment. This environment is created by hooking and replacing the runtime code of registry and file management functions in the target process' memory.

3.1.1 Considerations

There are two items that need to be considered before starting with the implementation of virtualized processes: Must the underlying logic be the same or different for unmanaged versus managed processes? And must the underlying logic be the same or different for a packaging process versus a normal process? A good application design should lead to a "No" as answer on both questions, but there are some insuperable issues that need to be solved first.

As documented in section 2.2.4 there is a difference between the injection procedure for an unmanaged process and for a managed process. But since, during runtime, managed code is in fact the same as native code with an extra layer on top, the underlying logic can stay the same. In code, there's only a difference in the injection procedure and in the injected library's initialization procedure. This difference is documented in the following sections.

A packaging process and a normal process are in essence the same. The main difference is that a packaging process is much more intensive on resources, it is constantly creating new resources while also opening many non-virtualized resources provided by the host operating system. This difference is surmountable by implementing a solid, flexible virtualized environment. Another minor difference is that the packager must be extended in order to be able to extract the main executable and other attributes from the package as soon as the packaging procedure has completed, the implementation of this extension is documented in the next section.

3.1.2 Implementation

The type of executable is determined by trying to extract an assembly manifest from the resource. A manifest is a text file containing metadata about .NET assemblies, and is thereby only extractable from managed resources. In other words, when the assembly manifest can't be extracted from the executable, the executable is expected to not use the .NET framework, meaning that the executable can be used to create and inject a process from without needing a workaround wrapper process.

As already discussed, a packaging process and a normal process are in essence the same except for the fact that a packaging process needs to be able to collect extra attributes during the application installation procedure. This is implemented by an extended file system provider, which watches the incoming messages and collects the necessary data by filtering the messages. When the installation procedure has successfully finished, the collected data is presented to the user who must then configure the fundamental configuration options before being able to use the packaged application.

3.1.3 Process Lifecycle

The lifecycle of a virtualized process can be divided into six phases. Each phase – as displayed in the following figure – is documented in this chapter's subsections.

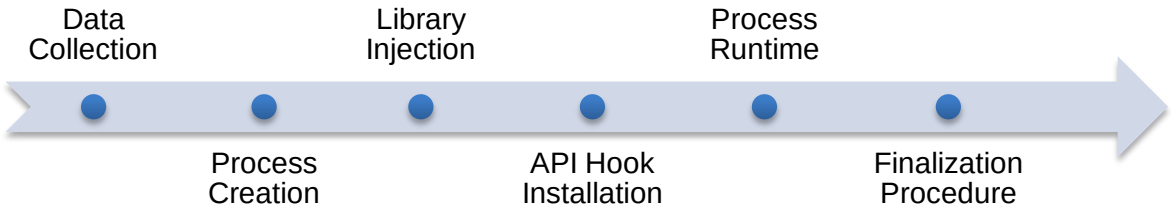


Fig. 4: Virtual Process - Lifecycle Overview

3.1.3.1 Phase 1: Data Collection

The data collection phase involves reading the packaged application's data file. By default, this data is contained in the "ApplicationData.xml"-file located in the root of the container. But remember, this is only a default value. The location of the data file can also be passed as a parameter to the AppStract host application, while the default value specified above is also configurable in the "AppConfig.xml"-file located in the root of the container.

This phase also involves verifying the retrieved data. First, the integrity of the data file is checked to avoid using corrupted data. Then, all resources specified in the data file are checked on existence and if possible also on data integrity.

3.1.3.2 Phase 2: Process Creation

Once the necessary data is collected and verified, the process can be started. The start-up procedure consists of two steps. First, EasyHook is prepared for usage. This involves the registration of all used libraries with the GAC, the creation of the channel that will be used for the IPC, and the preparation of the object that will be responsible for marshaling data across the remoting boundaries.

In the second step, the type of the assembly is determined and the required start-up procedure is executed. For unmanaged executables, a call to `CreateAndInject()` suffices. For managed executables, the procedure is more complicated since a wrapper process needs to be started. A wrapper process is a process that is able to run without altering any system resources. The injected library will then launch the packaged application inside the wrapper process during the fifth phase of the process's life cycle.

3.1.3.3 Phase 3: Library Injection

The library injection phase is completely handled by EasyHook. During this phase the `AppStract.Inject.dll` library is injected into the guest process. After injection, the entry point of the injected library initializes the server code inside the guest process. Initialized server code is able to communicate with the host process. This communication is handled by a communication bus which wraps an inter-process connection using the IPC channel created by the second phase.

3.1.3.4 Phase 4: API Hook Installation

When the library is injected and the server code is initialized, it's time to install the API hooks. During the initialization in phase 3 a registry of API hooks was built, this is the data source used by this phase. The API hook installation is completely handled by EasyHook. The server code only needs to provide a pointer to the function to hook, a replacement function, and a callback object.

3.1.3.5 Phase 5: Process Runtime

Now that all hooks are in place, the guest process is allowed to run. With native code, a wakeup call suffices. But with managed code, the entry point of the packaged application must still be loaded and launched into the guest process. This involves the use of .NET Reflection, an API that enables an application to dynamically discover information about classes in order to access class members, create new types at runtime, or modify its own behavior by calling methods at runtime using late-binding and dynamic invocation tech-

niques. By using .NET Reflection, the entry point of the executable is extracted and dynamically invoked with the parameters specified in the packaged application's data file.

During the process's runtime the user is able to use the portable application as if it is a normal, installed application.

3.1.3.6 Phase 6: Finalization Procedure

The process's finalization procedure involves of flushing all data from the guest process to the host process. A successful finalization includes the successful flushing of this data before the guest process exits and before the host process exits. This procedure can be invoked for several reasons:

- Planned process shutdown: The user decided to end the session.
- System shutdown: The operating system is shutting down.
- Guest receives the kill signal: The task manager kills the guest process.
- Host receives the kill signal: The task manager kills the host process.
- Guest process crashes: An unexpected exception occurred.
- Host process crashes: An unexpected exception occurred.
- Operating system stops: The OS crashes, the system looses power, …

This list is ordered on the severity of each case. A PLANNED PROCESS SHUTDOWN is the normal way of how a process exits. Such shutdown enables the finalization to be executed without extra concerns.

A SYSTEM SHUTDOWN sends an exit signal to all running processes. When this signal is sent, each process gets an undefined amount of time/clock cycles before the operating system exits. In Windows Vista the system shutdown might even be stalled or completely cancelled in case a process is not able to exit in the given time window. A shutdown of this kind is considered as safe, although it introduces some uncertainties.

A KILL SIGNAL always leads to a race against the clock. This is the case for the guest process and if the guest process owns data that is unknown to the host process, also for the host process. When the virtualized guest process receives the kill signal from the operating system, the process might still be able to immediately flush all new data to the host process, although this procedure is not assured to be infallible. When the host process receives the kill signal from the operating system, the process might be able to kill the guest process first and save the not-yet synchronized data.

A PROCESS CRASH will always cause a loss of unsynchronized data. A crash of the guest process means that all data, that has not been synchronized yet, is certainly lost. But on the other hand, the already synchronized data which might be kept in cache by the host

process is not lost. This data will immediately be written to the database files. A crash of the host process means that all data that has not been written to the database yet, is lost. Such crash also causes the guest process to crash because of a failure of the inter-process connection. In future versions of AppStract, this issue can be resolved by giving the guest process the ability to start a new host process which continues the work of the crashed host process. Although, a well developed application should never crash, so the proposed solution should be superfluous.

When the OPERATING SYSTEM STOPS unexpectedly, AppStract looses all non-saved data. This is completely normal behavior, since every application suffers from the same problem.

A loss of data might break the packaged application and even make it unusable. In future development of AppStract, a snapshot system should be introduced so changes in the package might be reverted in case the package is broken.

3.1.4 API Hooks

There is only one function related to process management that concerns AppStract, namely the `CreateProcess()` function. A call to this function creates a new process and the primary thread of the new process.

There are four types of relations that processes can have to a virtualized process:

- Parent process
- Sub-process
- Child process
- Other process

The parent process is the process that created the virtualized process. This will always be an instance of an AppStract server application like the AppStract Manager or the AppStract Host applications.

A sub-process is a process executed in parallel with the original process, comparable to how threads are executed in a process. Sub-processes are created when a process splits itself into multiple ‘daughter’ processes. API hooks installed in the original process are automatically installed in such sub-processes.

A child process is a process created by the virtualized process as a completely new and independent process. The creation of a child process occurs with for example .MSI installers, which creates both a server and a client process during runtime. Child processes

inherit most of their parent's attributes, but child processes don't inherit the installed API hooks. This is why a hook for `CreateProcess ()` must be installed. The implementation of this function creates the child process with all necessary API hooks installed. Although, there's one consideration that needs to be made, should the child process use the same resources/databases as the parent process or should all processes run in completely separate environments? In fact, resources should be shared as less as possible in order to improve performance. But with for example .MSI installers both processes must use the same resources, while this might not be the case with other processes. How can the code determine if two processes need to share their resources, or should child processes always share with their parent? Up till now, this is an unanswered question.

The 'other' category holds all other kind of relations, for example the processes running next to the virtualized process without any direct relation. This category does not affect the virtualized processes started by AppStract.

3.2 Registry

The Registry is introduced with Windows 95, to replace the old INI files. INIs are short text files with a basic structure, used to store configuration settings. INI files have many limitations including size restrictions, lack of internal organization and lack of support for multiple users and alternative hardware configurations. Sometimes, files using the INI file format will use a different extension, such as ".CFG", ".conf", or ".TXT". Since Windows 95, Microsoft began promoting the use of Windows Registry over the INI file. More recently, XML-base configuration files are gaining popularity. [13]

3.2.1 The Windows Registry – Introduction

Before being able to discuss the implementation of the virtual registry, the registry basics and its key terms need to be explained. These terms are documented in the next sections.

3.2.1.1 Logical Structure

The Registry contains two basic elements: keys and values. These are stored in a hierarchical database with a tree-like structure. The Registry can be compared to the File System; Keys are similar to folders, while values are similar to files. [16] In the Registry the hierarchy goes:

Key \\ *Subkey* \\ *Value*

Keys are named by backslash-delimited strings. Each key in the registry can have one or more values, which can contain strings, integral values, or binary data. Value names can contain backslashes, but doing so makes them difficult to distinguish from their key paths.

Registry Values are name/data pairs, stored within keys. Values are referenced separately from keys. There are a number of different types of values [15]:

ID	Name	Description
0	REG_NONE	No Type
1	REG_SZA	String value
2	REG_EXPAND_SZ	An expandable string value that can contain environment variables
3	REG_BINARY	Binary data (any arbitrary data)
4	REG_DWORD/REG_DWORD_LITTLE_ENDIAN	A DWORD value, a 32-bit unsigned integer *
5	REG_DWORD_BIG_ENDIAN	A DWORD value, a 32-bit unsigned integer *
6	REG_LINK	Symbolic link (UNICODE)
7	REG_MULTI_SZA	Multi-string value, an array of unique strings
8	REG_RESOURCE_LIST	Resource List
9	REG_FULL_RESOURCE_DESCRIPTOR	Resource Descriptor
10	REG_RESOURCE_REQUIREMENTS_LIST	Resource Requirements List
11	REG_QWORD/REG_QWORD_LITTLE_ENDIAN	A QWORD value, a 64-bit integer *

Table 1: Registry Value Types

** Endianness: In computing, endianness is the byte ordering used to represent some kind of data.*

Big Endian: The most significant byte (MSB) value is stored at the memory location with the lowest address. The other bytes follow in decreasing order of significance.

Little Endian: The least significant byte (LSB) value is stored at the memory location with the lowest address. The other bytes follow in increasing order of significance.

3.2.1.2 Hives and Root Keys

The Registry is split into seven logical sections, which are called hives. Each hive starts at a root key [11][12]. The following table is only a brief overview of the seven registry hives, more detailed information about the registry hives and their function can be found in Appendix 2.

Root Key	Contents Description
HKEY_CLASSES_ROOT	Information regarding OLE (Object Linking and Embedding)
HKEY_CURRENT_USER	User-specific information
HKEY_LOCAL_MACHINE	Global information pertaining to hardware and software
HKEY_USERS	User information
HKEY_CURRENT_CONFIG	Global information for the current hardware configuration
HKEY_DYN_DATA	Information of the current computer status (Windows 9x family)
HKEY_PERFORMANCE_DATA	Information of the current computer status (Windows NT family)

Table 2: Registry Root Keys

In reality, there are only two root keys, HKEY_LOCAL_MACHINE and HKEY_USERS. Documentation files sometimes reference to these two keys with abbreviations, namely HKLM and HKU. Three of the 'non-real' root keys are aliases, the Registry equivalent of Windows shortcuts, deriving their contents from subkeys of HKEY_LOCAL_MACHINE and HKEY_USERS. Any change in one key is reflected in its alias [12].

Alias	Branch of which the alias is derived
HKEY_CLASSES_ROOT	HKLM\Software\Classes
HKEY_CURRENT_USER	HKEY_USERS*%Current_User_SID%*
HKEY_CURRENT_CONFIG	HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current

Table 3: Registry Aliases

According to this table the alias HKEY_CLASSES_ROOT might seem redundant because it is always derived from the same branch. The reason that this alias is retained is to ensure compatibility with earlier versions of COM. The HKEY_CURRENT_USER is different for each user and is available to make it easier to work with the registry since reading and changing keys and values for the current user are one of the most frequent actions taken on the registry. As reflected in the table, HKEY_CURRENT_CONFIG is a pointer to HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. But this subkey and HKEY_CURRENT_CONFIG are in reality merely pointers to the content of the num-

bered Hardware Profile subkey used for the current hardware configuration [19]. These subkeys are stored under the same key as the Current subkey. The data for the current configuration can be viewed and edited in any of the three locations. The number of the hardware profile used for the particular session can be acquired by reading the value of HKLM\SYSTEM\CurrentControlSet\Control\IDConfigDB\CurrentConfig

The fourth 'non-real' root key, HKEY_DYN_DATA, is created on-the-fly from device status information each time the computer boots and is only used in Windows versions released before the NT kernel. Later versions of Windows derived from the NT kernel, use the HKEY_PERFORMANCE_DATA root key, which is the fifth 'non-real' root key. The data for this key is collected on-the-fly when it's request [13]. Both keys are also referred to as virtual keys [18].

3.2.1.2.1 Reserved keys

The Windows registry specifies seven reserved handles for the root keys [12], which are listed in the following table.

Key Name	Decimal Value	Hexadecimal Value
HKEY_CLASSES_ROOT	2147483648	0x80000000
HKEY_CURRENT_USER	2147483649	0x80000001
HKEY_LOCAL_MACHINE	2147483650	0x80000002
HKEY_USERS	2147483651	0x80000003
HKEY_PERFORMANCE_DATA	2147483652	0x80000004
HKEY_CURRENT_CONFIG	2147483653	0x80000005
HKEY_DYN_DATA	2147483654	0x80000006

Table 4: Registry Reserved Keys

These handles are always open and can't be closed. Almost all applications rely on these values for their initial access to the registry. This means that the virtual registry can't use these handles and should reserve seven handles of its own. Therefore the virtual registry reserves handles within the range of the hexadecimal values 0x00000001 'till 0x00000007. 0x00000000 can't be used because this value is reserved by the Windows operating system to identify null handles.

3.2.1.3 Class Identifiers

There's one group of subkeys feared by many programmers because of their complexity. These are the ActiveX class identifiers, known as CLSIDs. [15]

3.2.1.3.1 ActiveX

ActiveX is a Component Object Model (COM) developed by Microsoft for Windows. It's a set of rules for how applications should share information; it provides interaction between applications and within otherwise usually static applications, such as Web pages. [13]

3.2.1.3.2 CLSID

Each ActiveX class, whether it is a compound document object, an ActiveX control or some other class of ActiveX module, is assigned a unique CLSID. The unique string used for each CLSID is called a globally unique identifier, or GUID. GUIDs are used to provide a unique reference in any context.

CLSIDs are stored as 16-byte values and appear as long strings of numbers enclosed in braces. Each CLSID takes the same form of 8, 4, 4, 4 and 12 digits separated by hyphens. Windows stores CLSID-entries in `HKEY_CLASSES_ROOT\CLSID`.

Example of a CLSID: `{06B81C12-A5DA-340D-AFF7-FA1453FBC29A}`

GUIDs are randomly generated on application's request. Because of this random generation, GUIDs are not guaranteed to be unique. But the total number of unique keys (2^128) is so large that the probability of the same number being generated twice is negligible.

GUIDs representation can be little endian or big endian, so all APIs need to ensure that the correct data structure is used.

3.2.1.3.3 ProgID

Program Identifiers are aliases for CLSIDs [13][17]. Like the CLSID, the ProgID identifies a class but with less precision because it is not guaranteed to be globally unique.

ProgIDs are much easier for humans to handle, taking the form:

<vendor>.<component>.<version>

Vendor is sometimes replaced by the program name, an example is:

Word.Application.12

All ProgID entries in the Registry have a subkey called `CLSID`, which lists the associated class identifier. The associated CLSID can be searched in the registry to find the separate subkey providing much more detail about the actual properties of the class referenced by the ProgID alias.

3.2.2 Considerations

The implementation of the virtual registry can't be started without some important core questions being answered first. How to ensure data integrity between aliases? How to handle the Current User, and who is the Current User? How to handle the Current Config, is this the same for every computer? How to handle CLSID's? How to implement the different value types used in the registry? What about the reserved keys? How to avoid Windows' registry virtualization interference? …

Almost all of these questions have some kind of relation to root keys and/or registry aliases and their implementation, this implementation needs to be solid and reliable. The only two exceptions are "Who is the Current User?" and "What is the Current Config?"

3.2.3 Implementation

The implementation of the virtualized Windows registry can be seen as one of the most complex components of this thesis. The behavior of the Windows registry needs to be exactly copied to the virtualized registry, even the slightest difference might cause seemingly random crashes. This section will introduce some of the key terms of the virtual registry implementation.

3.2.3.1 AccessMechanism

Each incoming request gets a value of the `AccessMechanism` enumeration assigned. The following table provides an overview of this enumeration.

Member Name	Description
CreateAndCopy	Unknown keys and values are created in the virtual registry and copied from the host's registry.
TransparentRead	Read request for unknown keys and values are passed to the real registry, while write requests are always performed on the virtual registry.
Transparent	All requests are passed to the host's registry.

Table 5: AccessMechanism Enumeration

3.2.3.2 VirtualRegistry

`VirtualRegistry` is the class providing the registry logic. It holds two sub-types of registries, namely an instance of the `VirtualRegistryData` class and an instance of the `TransparentRegistry` class.

`VirtualRegistryData` is the class holding all known keys and values. The keys in this class all have a constant handle assigned, in other words keys from `VirtualRegistry-Data` don't have to be closed since the handles are not dynamically assigned. This class handles requests with the `AccessMechanism`'s `CreateAndCopy` or `TransparentRead` assigned to them. Although there are two exceptions with `TransparentRead`, when the key or value can't be found or when the requester needs to write, then the `Transparent-Registry` is used.

`TransparentRegistry` is a gateway to the real registry. It buffers key handles and links them to the real registry. These key handles must always be closed after usage, just as with the real registry.

3.2.3.3 Current User

In the implementation, the Current User is always "`S-APPSTRACT-User`", which is programmed as a constant. This constant is guaranteed to be unique because Windows uses SID's to identify users, and SID's are always formatted as "S-*[some numbers]*". When the process requests a by the virtual registry unknown key/value, the key/value is copied from the real registry to the virtual current user, thus a request for Current User gets the `CreateAndCopy` value assigned to it.

In the future, a registry editor might be developed for the virtual registry, giving the more advanced users the ability to manually edit the registry.

3.2.3.4 Current Config

All requests to the Current Config get the `TransparentRead` flag assigned to them. The Current Config is different for each computer and/or OS version. Read values from this hive should always reflect the state of the current host.

Requests for Current Config get the `TransparentRead` flag assigned and not the `Transparent` flag (which might seem more logical) because some poorly written applica-

tions might try to write to this location, even if it's discommended by Microsoft. Write actions on `HKEY_LOCAL_MACHINE`, `HKEY_CLASSES_ROOT`, or `HKEY_CURRENT_CONFIG` get automatically redirected by Windows' registry virtualization component. Allowing transparent write actions on any of these three hives would mean that AppStract leaks data to the host's registry.

3.2.3.5 Aliases

The Windows registry makes uses of aliases, as described in Chapter 3.1.1.2. This means an application might read from multiple locations while still receiving the same data, the same logic accounts for write actions.

Functionality like this requires a new layer of abstraction. This layer *translates* all registry-paths to one linking to a "*real*" registry hive. Doing this guarantees the data integrity between all alias hives.

3.2.3.6 Value Types

The Windows registry value types don't exist inside AppStract. In C# every value is an `object`, while in SQLite every value is a `blob`. The types are preserved for the external code/process by linking a value from the `ValueType` enumeration to the object or blob.

3.2.4 API Hooks

In the current implementation of AppStract five functions are hooked and implemented, in future versions three additional functions need to be added in order to ensure the stability of the virtualized registry.

The following is a list of the functions related to registry key functions [16]:

`RegCloseKey()`
: Closes a handle to the specified registry key.

`RegCreateKeyEx()`
: Creates the specified registry key.

~~`RegDeleteKey()`~~
: Deletes a subkey and its values.

`RegOpenKeyEx()`
: Opens the specified registry key.

~~`RegEnumKeyEx()`~~
: Enumerates the subkeys of the specified open registry key.

The following is a list of the functions related to registry value functions:

`RegSetValueEx()`
: Sets the data and type of a specified value under a registry key.

`RegQueryValueEx()`
: Retrieves the type and data for a specified value name associated with an open registry key.

~~`RegDeleteKeyValue()`~~
: Removes a named value from the specified registry key.

The registry API specifies many other functions. These functions are not hooked since all of them are extensions for the functions listed above.

Due to lack of time, three of the eight fundamental functions have not been implemented:

- `RegDeleteKey()`
- `RegDeleteKeyValue()`
- `RegEnumKeyEx()`

The first two functions are both delete functions and aren't really that important, most applications are able to stay usable without these. While the enumeration function `RegEnumKeyEx()` is much more important although still not indispensable, this last function is quite hard and time consuming to implement and is therefore postponed to a future version of AppStract.

3.2.5 Issues

The virtual registry is a core component of AppStract and fundamental for the usability of the AppStract application. Sadly, some of the key functions needed for the virtual registry are not implemented due to lack of time. Because these functions - and especially the `RegEnumKeyEx()` function - are unimplemented, it's very hard to track existing bugs down. One of these bugs is for example the issue of guest processes trying to create or open subkeys for unknown key handles. The most logical explanation for this issue would be that the unknown key handles are acquired by a call to the `RegEnumKeyEx()` function.

Another issue with the virtual registry is caused by unimplemented behavior in the `RegQueryValueEx()` function. It is documented by Microsoft `[16]` that the function retrieves the type and data for the keys unnamed or default value if the `lpValueName` parameter is a null pointer. While in the current implementation no data is returned, telling the guest process that the specific key doesn't have an unnamed or default value.

The only way those two known issues can be fixed, is by spending more time on the development of the virtual registry. Both issues are caused by unimplemented features that are in fact fundamental for the functionality of the virtual registry. The implementation of the `RegEnumKeyEx()` function is expected to be time consuming, since developing an enumeration system in non-object-oriented code isn't as easy as developing such system in object-oriented code. Also, the virtual registry isn't guaranteed to know all of the subkeys of the specific enumerated key. Implementing `RegEnumKeyEx()` would require some kind of transparency, combining subkeys from both the virtual and the real registry. All of this functionality must be coded in a way that makes it very efficient to be executed, with performance close to what's offered by the real `RegEnumKeyEx()` function.

It's expected that the virtual registry hides more bugs and issues. But these can't be tracked down right now, because any guest process is expected to crash in its initialization procedure on the issues described above. As a consequence, only three or four calls are made to the virtual registry. In order to be able to effectively debug the virtual registry, the component must be stressed by having to handle many more calls in a short amount of time. These calls must also include both transparent as non-transparent requests, representing a real world situation.

3.3 File System

3.3.1 Considerations

There are two ways to implement the file system. The implementation can be either a fully virtualized file system, or it can be a redirecting system. Developing a fully virtualized file system means that a system like NTFS or FAT must be implemented together with all file management functions as defined by the Windows OS.

Developing a redirecting system is much simpler than a fully virtualized file system. Basically, a redirecting system is nothing more than a switch which either passes the request to the real file system or either redirects the request to the application container. For example, a request for "`C:\Program Files\MyApplication\somefile.dat`" can be redirected to "`F:\Portables\MyApplication\Data\somefile.dat`" without the application noticing this.

3.3.2 Implementation

After considering both possibilities I opted for the redirecting system because of its efficiency and simplicity.

The main component of the redirecting file system is a file table, which is implemented with an `IDictionary<string, string>`. This can be seen as a two column table where the first column contains keys, and the second column contains values. The keys are the paths as requested by the application, while the values are the replacement paths. All paths are stored relative to the process's execution location to ensure portability.

3.3.3 API Hooks

In the current implementation of AppStract five file management functions are important to provide a virtualized file system [22].

`CreateFile()`
: Creates or opens a file, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, or named pipe.

`CreateDirectory()`
: Creates a new directory.

`DeleteFile()`
: Deletes an existing file or existing directory.
NOTE: *It is not documented that this directory must be empty.*

`LoadLibraryEx()`
: Loads the specified module into the address space of the calling process.

`RemoveDirectory()`
: Deletes an existing empty directory.

The `CreateFile()` function is implemented by Microsoft to create both files and directories, and also to open both files and directories. The `CreateDirectory()` function's only functionality is to pass calls to the `CreateFile()` function. Nevertheless, AppStract also implements the `CreateDirectory()` function in order to know the kind of resource requested by the guest process. In AppStract, calls to `CreateFile()` are intercepted only for creating or opening a file or directory. All requests for physical disks, volumes, console buffers, tape drives, communications resources, mailslots, and named pipes are transparently passed to the underlying host operating system.

Note that the difference in functionality of `DeleteFile()` and `RemoveDirectory()` is similar to the difference in functionality of `CreateFile()` and `CreateDirectory()`. The `DeleteFile()` function has the same, and more, functionality than the `RemoveDirectory()` function. The `RemoveDirectory()` function's only functionality is to pass calls to `DeleteFile()`, this is why `RemoveDirectory()` is not hooked nor implemented by AppStract.

The Windows file management API specifies an `OpenFile()` function. The existence of this function is necessary in order to support legacy software which is developed for 16 bit versions of Windows. This functionality doesn't have to be implemented by AppStract because it is expected that 32 bit versions of Windows will internally redirect a call to `OpenFile()` to a call to `CreateFile()`. If this assumption is not correct, AppStract will not provide support for legacy applications, which should in reality be no big deal for any user.

3.3.4 Issues

When developing the file system I noticed that a redirecting system is not compatible with I/O to pipes, network shares and removable devices like USB storage or DVD's. This is quite easy to fix by placing a filter between the implementation of the file system and its interface. A second, more severe, issue is that all file access is redirected. This is no problem for file access performed by the applications code. But when the user wants to save a file in for example his desktop directory, he expects it to be saved there while in reality it will be saved to a location in the application's container.

There are two possible fixes for this last issue. The first possibility would be that we use a more dynamic file system, which stops redirecting after a specified period of time. As an example, most applications boot within 15 seconds and it will probably only need files during this first 15 seconds. So when the application is running the file table can be released. In my opinion this "solution" contains too many probabilities, and probabilities are never acceptable when it comes to computing. The fact that probabilities are a no go in computing makes this solution inacceptable.

Another solution would be to keep using the redirecting system during the whole lifetime of the process, meaning all files will always be read from and saved to the container file. The user can then export all saved files using an extra tool, provided by AppStract. This solution is less transparent for the user, but way more reliable when it comes to the application's functionality. It also enables the file system to be wrapped with extra layers of security and privacy. The security component is present because potentially system endangering actions are unable to be executed outside of the virtual file system. And the privacy component can be implemented by allowing system administrators to configure the type of files that are allowed to be exported. These allowed or disallowed types can be made configurable by looking at specific file attributes.

3.4 Container File

The container file is the file containing all application data and settings together with the AppStract server application.

3.4.1 Considerations

The primary requirement of the container file is that the Windows operating system must have transparent access to it, while the user must be able to execute the file as a normal executable. A secondary requirement would be that the container file is inaccessible to browse for users. The container file should only be accessible by associated container management utilities delivered with AppStract and during runtime only by direct calls from the AppStract host and AppStract guest processes.

3.4.2 Implementation

The container file is an unimplemented feature due to lack of time. Currently, the container is a directory and the user needs to manually start the server from inside the container directory. In future development this implementation can be easily replaced by the implementation as in the next paragraph.

The general idea behind the implementation was to use the zip file format for the container. The executability of the container could then be implemented by implementing functionality similar to how self-executable archives are implemented. The self-executing functionality can be replaced by a parameterized link to the server's executable inside the container. The Windows operating system is then able to transparently browse the container, using deep links, and to execute the container, using a link to the container's root executable file.

4 CONCLUSION

This thesis documents an in-depth research of how application virtualization can be designed and implemented in a managed programming language. The fundamental parts of this design are the virtual registry and the virtual file system. The implementation of this thesis still contains many bugs, although none of these are caused by the design as described by this thesis. This proves that the proposed design is able to function in a real implementation, and it can hopefully be a strong foundation for an open source alternative for the existing commercial implementations [Appendix 3].

The implementation of the interception of function calls is one of the most technical parts of this thesis. In order to understand how function calls can be intercepted one must know how Windows manages its libraries and what possibilities Windows offers to implement function call interception. One must also understand the principles of code injection and the dangers that come with it. But due to the availability of the EasyHook library, the implementation of this functionality is elevated to a higher code level making it relatively easy to develop.

Another very technical part is the virtual registry. The way the Windows registry functions is fairly complicated and full of strangely connected constructions. Such constructions mostly have a historical origin, and are implemented in order to support legacy software packages. This thesis tries to implement an exact copy of the behavior of the Windows registry. Although, this thesis reveals that implementing an exact copy of the Windows registry is close to impossible. If there would have been more time to complete this thesis, the virtual registry would have functioned well enough for many applications. But in the current implementation there is no virtual substitute for some of the fundamental functions, making the virtual registry unusable for many applications. On the other hand, this thesis should provide all necessary theoretical knowledge for developing a virtual registry. It is only the practical implementation that doesn't suffice for daily use.

The implementation of the virtual file system was relatively easy when compared to the other parts of this thesis. This is because the virtual file system doesn't really provide a virtual environment. Providing such environment would have required the design of a new kind of file system, on the technical level of file systems like NTFS and FAT. In the current implementation, the file system is virtualization makes use of the trampoline function functionality as provided by the Detours library. When a function call is intercepted, the para-

meters are altered and passed to the trampoline function. By altering the parameters, the requested paths are redirected to the real location of the resources.

Application virtualization is the fastest growing segment of virtualization technologies [23], and this thesis might be a strong foundation for an open source alternative for the existing commercial implementations [Appendix 3]. The way of how such system can be implemented is completely documented by this thesis, and the fundamental techniques are fully tested by the implementation of the AppStract application. The full source code of this application can be found on the CD attached to this thesis, or on the internet [http://code.google.com/p/appstract]. Although, AppStract will probably never be a full replacement alternative for the commercial alternatives, since these alternatives are always bundled in a package containing applications for server virtualization, desktop virtualization, and application virtualization, which all can be managed with a single virtualization management application. But still, AppStract is a nice research project suitable to take your first steps in the amazing world of virtualization.

REFERENCES

[1] http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.80).aspx
- *.NET Framework Remoting Overview*

[2] http://msdn.microsoft.com/en-us/library/ms973857.aspx
- *Microsoft .NET Remoting: A Technical Overview*

[3] http://www.codeproject.com/KB/WCF/net_remoting.aspx
- *All you need to know about .Net Remoting*

[4] http://www.codeproject.com/KB/IP/Net_Remoting.aspx
- *NET Remoting with an easy example*

[5] http://www.codeproject.com/KB/IP/Remoting_Architecture.aspx
- *Remoting Architecture in .NET*

[6] http://msdn.microsoft.com/en-us/library/hk5f40ct.aspx
- *Assemblies in the Common Language Runtime*

[7] http://www.anandtech.com/IT/showdoc.aspx?i=3237
- *Application Virtualization – Business Use*

[8] http://en.wikipedia.org/wiki/Application_virtualization
- *Application Virtualization - Introduction*

[9] http://research.microsoft.com/en-us/projects/detours/
- *Introduction to Microsoft Detours*

[10] http://research.microsoft.com/pubs/68568/huntusenixnt99.pdf
- *Detours: Binary interception of Win32 functions*

[11] http://www.codeproject.com/KB/dotnet/EasyHook64.aspx
- *EasyHook - The reinvention of Windows API hooking*

[12] http://www.easydesksoftware.com/rworks.htm
- *How the Windows Registry Works*

[13] http://mc-computing.com/Languages/RegistryIO.htm
- *Window Registry I/O*

[14] http://www.geekgirls.com/windows_registry01.htm
- *Understanding the Windows 9x Registry: Part I*

[15] http://www.geekgirls.com/windows_registry02.htm
- *Understanding the Windows 9x Registry: Part II*

[16] http://msdn.microsoft.com/en-us/library/ms724875(VS.85).aspx
- *Registry Functions*

[17] http://msdn.microsoft.com/en-us/library/ms688386.aspx
- *Function: CLSIDFromProgID*

[18] http://www.ddj.com/database/184416281
- *Understanding NT*

[19] http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/regentry/69675.mspx?mfr=true
- *HKEY_CURRENT_CONFIG*

[20] http://msdn.microsoft.com/en-us/library/aa965884(VS.85).aspx
- *Registry Virtualization*

[21] http://msdn.microsoft.com/en-us/library/ms724182(VS.85).aspx
- *About the registry*

[22] http://msdn.microsoft.com/en-us/library/aa364232(VS.85).aspx
- *File Management Functions*

[23] http://www.itrportal.com/absolutenm/templates/article-netwireless.aspx?articleid=5358&zoneid=58
- *Application Virtualization is fastest growing segment*

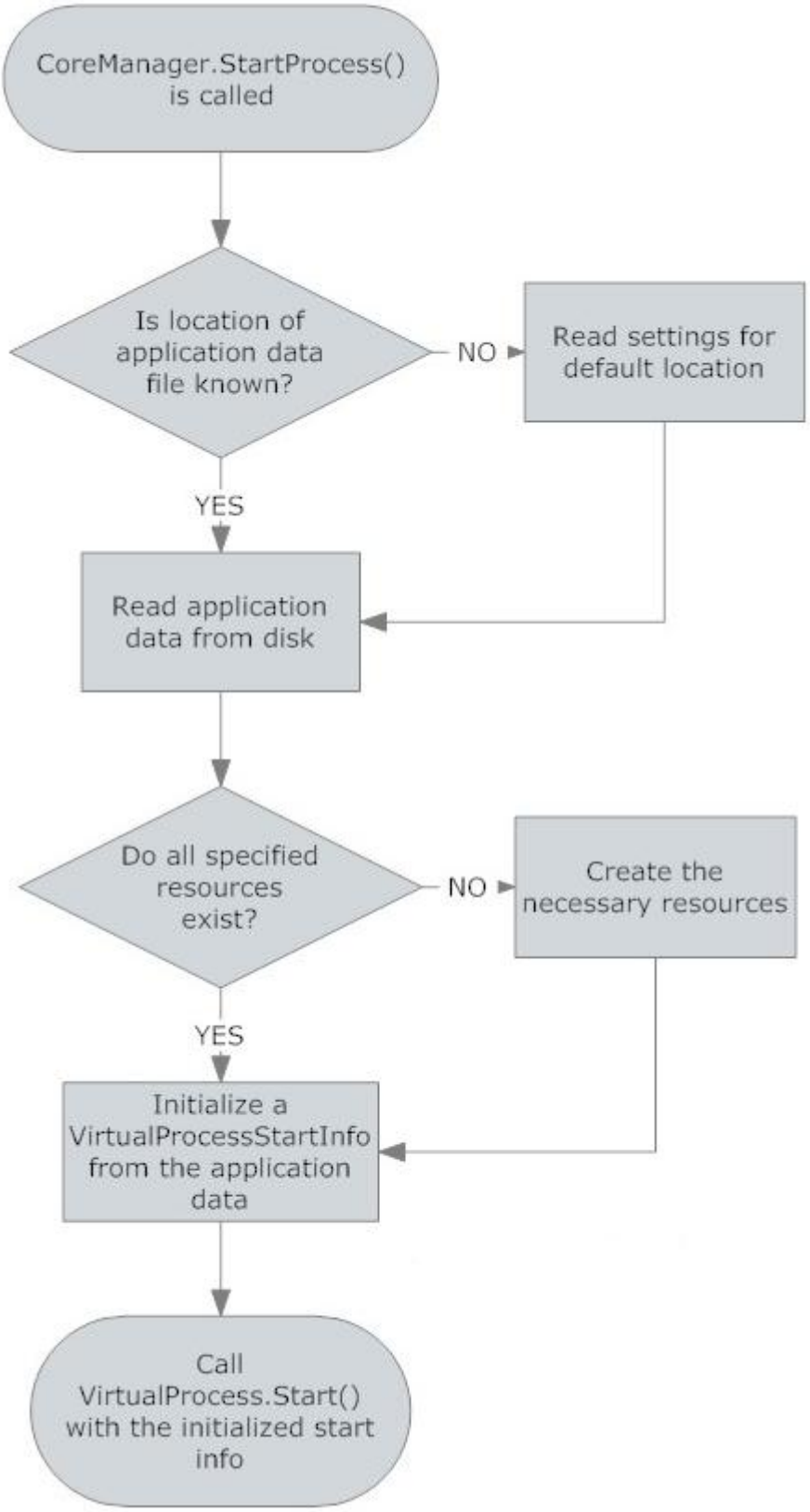
[24] EasyHook Documentation files
- *Documentation of the EasyHook library*

[25] http://msdn.microsoft.com/en-us/library/bb432182(VS.85).aspx
- *Application Compatibility Database*

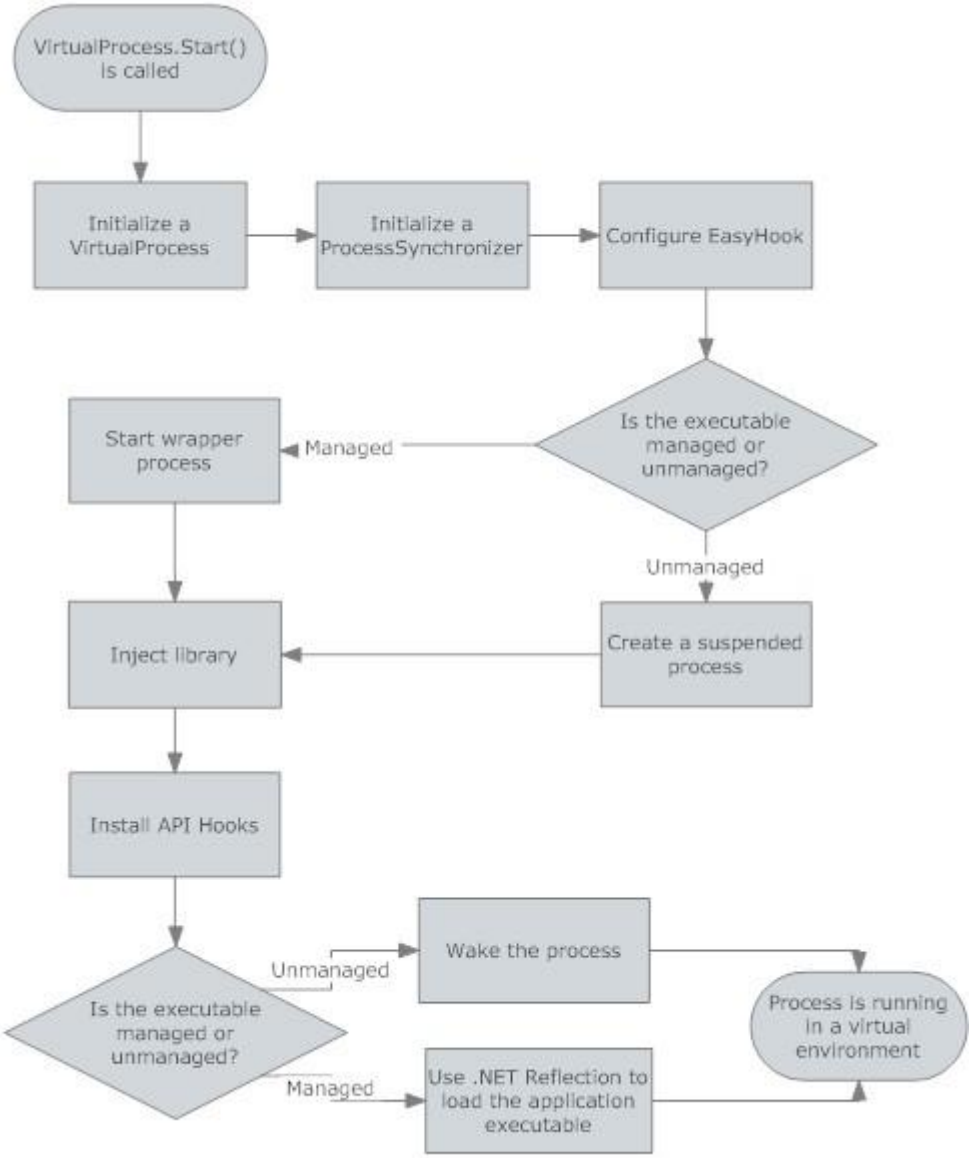
[26] http://msdn.microsoft.com/en-us/library/ ms682599(VS.85).aspx
- *Dynamic-Link Library Functions*

Flow Charts

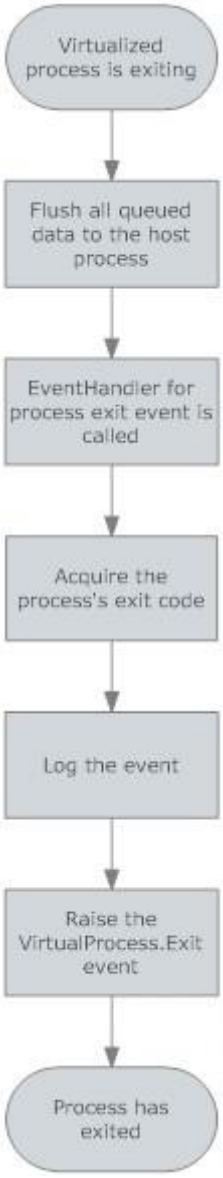
Process Start-Up Procedure – Phase 1



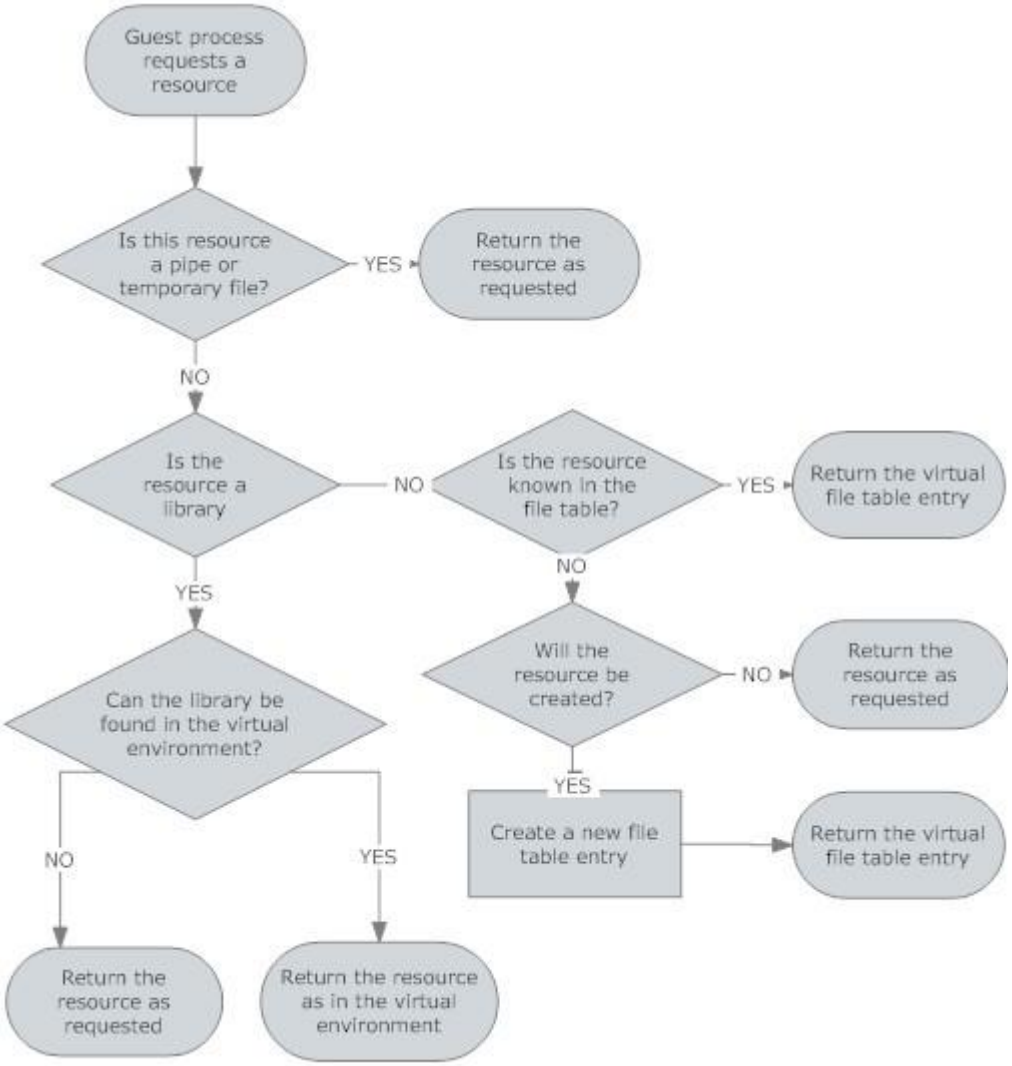
Process Start-Up Procedure – Phase 2



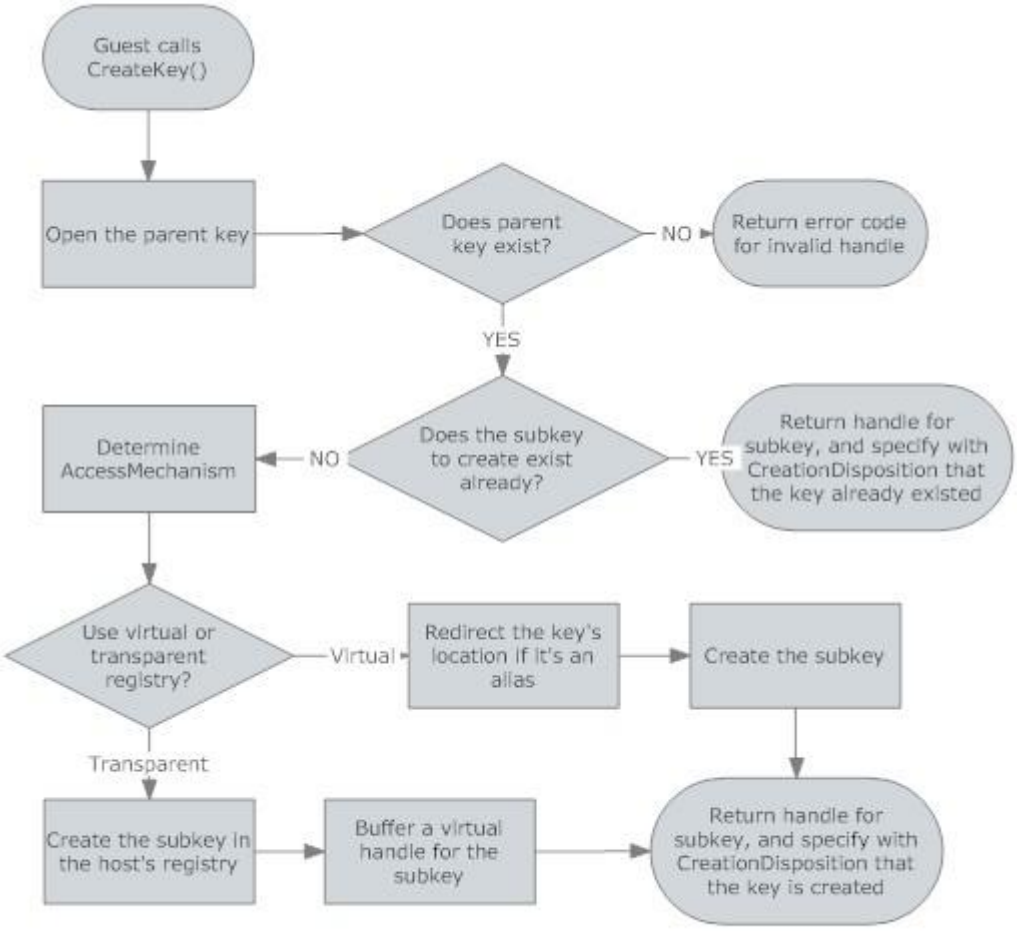
Process Shutdown Procedure



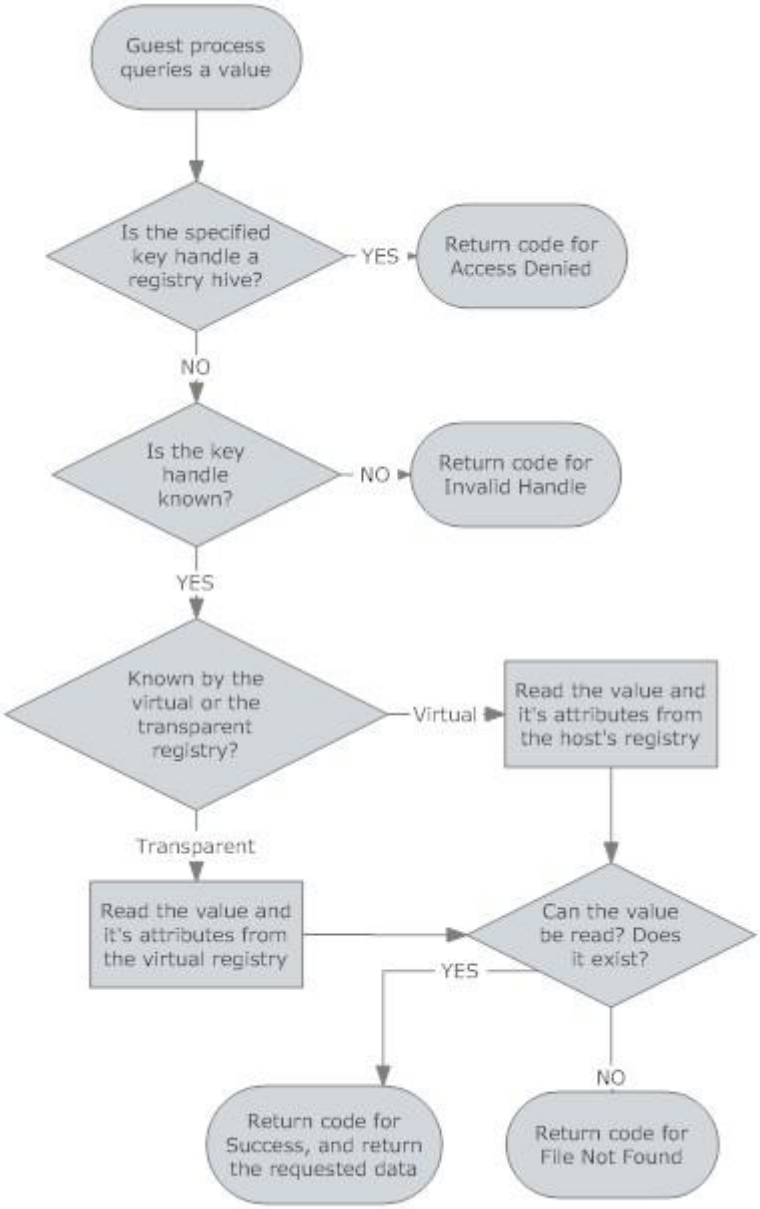
File System Logic Overview



Registry Logic Overview – Keys



Registry Logic Overview – Values



Registry Hives

HKEY_CLASSES_ROOT HKCR

- Contains information regarding OLE (Object Linking and Embedding), file association mappings, Windows shortcuts and some other aspects of the user interface.
- HKEY_CLASSES_ROOT contains the names of all registered file types and their associated properties.
- This key is a pointer to another subkey: KEY_LOCAL_MACHINE\Software\Classes

HKEY_CURRENT_USER HKCU

- Contains user-specific information.
- The content is derived from information in the HKEY_USERS key at logon time.

HKEY_LOCAL_MACHINE HKLM

- Contains global information pertaining to system hardware and applications software settings. The information and settings apply to all users who log on to the computer.
- This is a read-only system location, but poorly written applications may still try to write to this hive.

HKEY_USERS HKU

- Contains user information such as default application settings, desktop configuration and so on.
- Some of the information is user-specific and some is available to all users of the computer.
- The default, generic settings are stored in a subkey HKEY_USERS\.DEFAULT.
- A new subkey is created for each user who logs onto the computer. At first, that new subkey contains a copy of the contents of the .DEFAULT subkey. As the user changes desktop and application settings, those changes are stored in the user's subkey.

HKEY_CURRENT_CONFIG HKCC

- Handles Plug & Play settings and information about multiple hardware configurations.
- Settings in this key are derived from subkeys of HKEY_LOCAL_MACHINE.

HKEY_DYN_DATA HKDD

- Maintains a dynamic record of the current status of the computer.
- Only used in Windows 9x

HKEY_PERFORMANCE_DATA HKPD

- The performance data is not actually stored in the registry database. Instead, calling the registry functions causes the system to collect the data from the appropriate performance counter provider.
- Used in the Windows NT family.

Similar Products

VMWare ThinApp Application Virtualization

"Package once, deploy everywhere with VMware ThinApp. Run virtually any application on any device across any deployment medium, with no conflicts."

http://www.thinstall.com

Symantec Endpoint Virtualization Suite

"Symantec Endpoint Virtualization Suite (formerly known as Software Virtualization Solution Professional) is a revolutionary approach to optimizing software management. It serves the needs of traditional, virtual and hybrid enterprise endpoints, providing high productivity with controlled, guaranteed access to any Windows application. Application streaming provides the on-demand delivery mechanism and centralized license management, while virtualization places applications and data into managed units."

http://www.symantec.com/business/endpoint-virtualization-suite

Sandboxie

"Tired of dealing with rogue software, spyware and malware?
Spent too many hours removing unsolicited software?
Worried about clicking unfamiliar Web links?
→ Sandboxie runs your programs in an isolated space which prevents them from making permanent changes to other programs and data in your computer."

http://www.sandboxie.com

App-V (Microsoft Application Virtualization)

"Microsoft Application Virtualization (App-V) version 4.5 has been released to manufacturing. App-V 4.5 is the first Microsoft-branded release of the product formerly known as SoftGrid. It includes new capabilities designed to help IT support large-scale virtualization implementations across many sites and provides multiple delivery options, including over-the-Internet application availability to meet your business needs."

http://www.microsoft.com/systemcenter/appv/default.mspx

External Libraries

EasyHook

- *"The reinvention of Windows API Hooking"*
- Website: http://www.codeplex.com/easyhook
- License: GNU Library General Public License (LGPL)
    - http://www.codeplex.com/easyhook/license

SQLite

- *"Self-contained, serverless, zero-configuration, transactional SQL database engine"*
- Website: http://www.sqlite.org
- ADO .NET Provider: http://sourceforge.net/projects/sqlite-dotnet2
- License: Public Domain
    - http://www.sqlite.org/copyright.html