UNIVERSITY OF WATERLOO
Faculty of Engineering
Nanotechnology Engineering

Cross-Platform Application Development using .NET and Mono

Safe Software Inc.
Surrey, BC

Prepared By:

Rajesh Kumar Swaminathan
2A Nanotechnology
ID #20194189
rajesh@meetrajesh.com

May 6, 2007

Rajesh Kumar Swaminathan

#27-8289 121A St.

Surrey, BC V3W 1G6

May 6, 2007

Dr. Marios Ioannidis, Director

Nanotechnology Engineering

University of Waterloo

Waterloo, Ontario N2L 3B9

Dear Dr. Ioannidis,

This report, entitled, "Cross-Platform Application Development using .NET and Mono" is my second work term report for the term immediately following 2A spanning the months of January to April 2007. This report was completed during my work term at Safe Software Inc., a company that specializes in creating software for *geospatial* data manipulation. The purpose of this report is to investigate what is involved in building cross-platform software using technologies such as .NET and Mono, and to analyze each technology's pros and cons.

The idea for this topic was hinted to me by two people within the company: Dale Lutz, Vice President Technology and Graeme Hiebert, Senior Developer. Dale was very interested in Mono as a solution to building truly cross-platform software. Graeme, also the platforms team-lead, saw Mono as an alternative that would make it very easy for developers to port applications between multiple platforms, architectures and operating systems.

The topic of this report proved itself to not only be insightful and exciting, but a topic that, I hope, will be of immense usefulness to our company and the software product we ship. There are a lot of issues, benefits and outcomes to be discussed while writing software that runs identically between various platforms; however, due to the size limitations of this report, I have restricted myself to a select few targeted topics that are of direct relevance to our applications design.

I vouch to the fact that I have received no further help other than what is mentioned above and in the references section in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Rajesh Swaminathan

20194189

Contributions

At Safe Software Inc., I was a platforms engineer within a medium-sized team of about seventy to seventy-five people. Safe Software specializes in writing software for *geospatial* data manipulation. My main responsibility was to port our flagship software known as FME (Feature Manipulation Engine) to other platforms such as the UNIX platform and the 64-bit machine architecture.

Our primary goal for the four-month term was to push a major 2007 release. This release introduces major features and functionality additions to our software.

As platforms engineer at Safe, I ensured that our flagship software product, FME, was able to compile and run on Linux, Solaris, HP-UX, AIX, and 64-bit Windows operating systems. I was also responsible that our test-suite, comprising of about 2500 tests, ran successfully to an acceptable extent on these platforms. I worked closely with my team lead, Graeme Hiebert, in getting regular UNIX builds out onto our FTP servers whenever our clients would request them.

The porting process was fairly involved and tedious. Multiple project configurations had to be created which made maintenance quite difficult. In that light, I looked at various alternatives to ease the porting process. One such alternative was .NET, a development framework by Microsoft, released in early 2002. .NET can be compiled and run on UNIX-like systems using Mono, an open-source project sponsored by Novell. An easier porting process would mean an easier development process on non-standard platforms. This, in turn, would mean much more frequent builds available on our FTP server for clients.

Summary

The purpose of this report is to investigate what is involved in building cross-platform software using technologies such as .NET (pronounced dot-net) and Mono. The report aims to analyze each technology's pros and cons, and to study any trade-offs developers may need to make while considering .NET and Mono as a viable solution.

The Mono Website [1] was used as the primary source of information in conducting this analysis. In general, the report has been approached from the stand-point of a consulting engineer who is investigating the viability of .NET and Mono to developing cross-platform software solutions and is supplying a set of engineering recommendations to resolve any issues or problems that may arise upon a decision to choose and implement .NET/Mono.

The report begins with an introduction to .NET and Mono. Section 1 explains how .NET and Mono work, how they can be useful, and how they lead to a development model that is different from that of traditional compile-and-run softwares. The introduction also contains a basic idea of software portability and why it is important.

Section 2 discusses the various features, advantages, and benefits to be gained by using .NET and Mono. These benefits are addressed from a variety of multiple view-points including added business value, security, and internationalization.

Section 3 looks at common porting issues and their corresponding solutions that a developer might encounter by choosing .NET and Mono to develop cross-platform software. This sections also recommends best practices to avoid future problems and headaches.

Finally, Section 4 formulates a generic porting strategy that works for most software projects that rely on .NET and Mono. Because this porting strategy has been intentionally kept fairly generic, each organization's porting strategy may differ from the one outlined here. The porting strategy is only a guideline and is meant to be useful only as starting material.

Conclusions

The conclusion of the report is that technologies such as .NET and Mono are tremendously useful in shortening the porting process significantly. This results in an easier development process which leads to more frequent beta versions of software available on FTP for UNIX versions. The more frequent the software company puts up betas, the quicker they get feedback from clients, and the faster problems and bugs are resolved.

.NET and Mono provide tools and utilities that help shorten the porting process quite noticeably. They also make possible a "compile-once, run everywhere" kind-of environment whereby code can be compiled once on any operating system, copied over to another operating system, and made to run natively on that operating system without the help of any emulators. This is all possible thanks to Mono, an open-source project sponsored by Novell, which is an effort that aims to port the .NET framework to UNIX. Semi-compiled byte-code can be executed on any platform for which a port of Mono is available. Even at the time of writing, all major operating systems and machine architectures (including 64-bit) have been covered.

By moving to .NET, it is possible to run graphical interfaces on any platform natively without having to maintain multiple versions of the same code. Currently our graphical utility software known as "Workbench" runs only on the Windows operating system because it uses proprietary Delphi code. Converting this to .NET-comapatible code will enable users on any operating system like Linux, Solaris, Mac OS X, etc. to use Workbench natively.

.NET also allows developers to choose any language of their choice. This makes hiring developers easier as they are not required to know any one specific development language.

In conclusion, as long as developers have a clear porting strategy and constantly keep in mind the recurring issues, the .NET/Mono combination is an excellent solution for developing truly cross-platform software.

Recommendations

Based on the analysis and conclusions in this report, the following recommendations are proposed.

1. Because our current software is written in C++ and does not directly target the .NET framework, we would require a significant effort on part of the platforms team to port the current code-base to a state where it can be compiled by the .NET compiler. This requires a team of 4-5 full-time developers and is expected to take at least 2-3 years given the size of our current code-base.

2. Once the initial work has been done upfront, we would still need at minimum one full-time developer who will ensure that any new code written is tested on the various UNIX platforms. His sole responsibility will be to take care of minor porting issues such as case sensitivity, file paths, etc. These issues are common enough to warrant a full-time developer.

Table of Contents

List of Figures

List of Tables

1 Introduction

Most software written today for production use needs to be able to run on multiple platforms. The functionality provided by the software application must be identical regardless of the operating system in use, the hardware specifications of the host computer, or the platform or architecture of the machine. Further, if the software is GUI-based, the look and feel should not vary either. Even if variations are unavoidable, the goal is to minimize the effect of these variations as much as possible.

.NET, an application framework by Microsoft offers interesting promises in this direction. Mono, an open-source project sponsored by Novell, began an initiative about three years ago to port the .NET application framework to other platforms and operating systems, notably Linux. Since this project is open-source, it allows room to be further ported to other not-so-major platforms as well.

.NET, together with Mono, software developers can now truly imagine "write once, run everywhere" type applications where code is written once, but can be run on any system without recompilation regardless of platform, architecture or operating system.

1.1 What is .NET?

.NET is an "umbrella" term used to encapsulate a wide variety of products and technologies from Microsoft. In short, .NET is a collection of tools, utilities and components that aid developers in building console, GUI, and web applications, abstracting much of the lower level detail involved in building high-performance production software. Of these, the most important components are:

- **The Microsoft .NET Framework**, a collection of core classes and libraries to perform common tasks.
- **The C# programming language**, an object-oriented compiled language. What we're really talking about here is the C# compiler which compiles C# programs to MSIL (Microsoft Intermediate Language).

- **The Common Language Runtime (CLR)**, an MSIL interpreter and just-in-time (JIT) runtime that converts MSIL to native machine code.
- **ADO.NET**, a data access library.

1.2 What is Mono?

"Mono provides the necessary software to develop and run .NET client and server applications on Linux, Solaris, Mac OS X, Windows, and UNIX." [1] The project's objective is to completely port the Microsoft .NET development platform to UNIX, thereby allowing UNIX developers to build and deploy truly *cross-platform* .NET applications regardless of operating system and machine architecture. The project, although sponsored by Novell, is open-source. The current version of Mono, Mono 1.2, supports most of the common functionality offered within the .NET environment. Mono contains the core development libraries, as well as the development and deployment tools. At the time of writing, Mono's API coverage is limited to the .NET 1.1 API, with "spotty" support for 2.0. [2]

Mono has support for both 32- and 64-bit systems on a number of architectures as well as a number of operating systems. Officially supported operating systems are:

- Linux
- Mac OS X
- Sun Solaris
- BSD – OpenBSD, FreeBSD, NetBSD
- Microsoft Windows

Mono also has support for a wide variety of machine architectures, some of which are listed in Table 1 along with their corresponding runtime (JIT or interpreter) and operating system.

Mono contains the following components [2]:

Table 1: A list of supported architectures and their corresponding runtimes and operating systems. [4]

Architecture	**Runtime**	**Operating System**
s390, s390x (32 and 64 bits)	JIT	Linux
SPARC (32)	JIT	Solaris, Linux
PowerPC	JIT	Linux, Mac OSX
x86	JIT	Linux, FreeBSD, OpenBSD, NetBSD, Microsoft Windows, Solaris, OS X
x86-64: AMD64 and EM64T (64 bit)	JIT	Linux, Solaris
IA64 Itanium2 (64 bit)	JIT	Linux
ARM: little and big endian	JIT	Linux
Alpha	JIT	Linux
MIPS	JIT	Linux

- A Common Language Infrastructure (CLI) virtual machine that contains a class loader, a just-in-time compiler, and a garbage collector.
- A class library that can work with any language which works on the CLR. Both .NET compatible class libraries as well as Mono-provided class libraries are included.
- A compiler for the C# language. Future work on other compilers that target the Common Language Runtime are planned.

A summary of Mono's architecture can be seen in Figure 1.

1.3 What is Portability?

Portability, in computer science, simply means that an application written once can be run on any system or machine regardless of operating system, environment, file-system, CPU architecture, and machine platform. Such an application is known to be "portable".

Often, software applications are not portable immediately. They will need to be adapted to support different platforms, and occasionally, certain functionality will need to be dropped for certain platforms because they are simply not portable. Third-party libraries with no source code available are not portable because developers cannot change the code to adapt to various platform specifics.

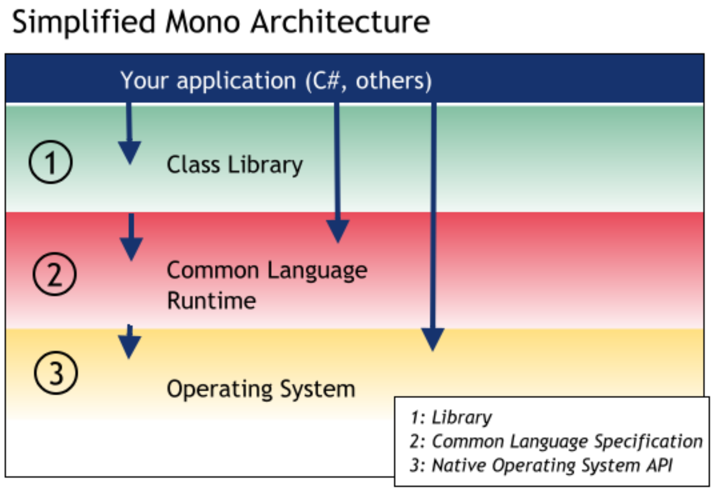
Simplified Mono Architecture
Your application (C#, others)
1
Class Library
2
Common Language Runtime
3
Operating System
1: Library
2: Common Language Specification
3: Native Operating System API

Figure 1: The Mono architecture diagram shows the sequence of steps involved in the execution of a C# program. [4]

This collective process of adapting software to run in environments different from the one for which it was originally designed is called “porting”.

There are many libraries available to deal with porting issue for things like socket programming and GUI development, but for the most part, developers always like to reduce the amount of work needed to be done to make their software portable. The .NET/Mono combination is interesting because it allows developers to write code that can at least be compiled anywhere and be run wherever Mono runs. [5] As seen before, Mono supports a wide variety of operating systems and system architectures, so there should be very minimal porting issues to take care of.

1.4 Recent Trends

The .NET/Mono combination moves away from a “write once, compile anywhere” paradigm to a “compile once, run anywhere” paradigm. Although the end goal is the same (the software must run seamlessly anywhere), the approach taken is entirely different. The “compile once, run anywhere” paradigm implements cross-platform compatibility at the compiled binary level, rather than at the source code level.

To illustrate the difference, consider an application written in C++. The application, compiled on Linux, can only run on Linux. It cannot be run on Windows. However C++ compilers do exist on Windows (like Microsoft's VC++). Thus in order to run the application on Windows, we would have to re-compile the application with a Windows-based compiler.

However, an application written in C#, can be compiled anywhere, either on Windows or on UNIX. The compilation does not actually compile the code down to the binary-level. Instead, it compiles it into an intermediate language called Microsoft Intermediate Language, or MSIL (See Figure 2 for detail). Upon execution, the CLR (common language runtime) translates the MSIL to machine code specific to the host machine's platform specifics, and then executes it.

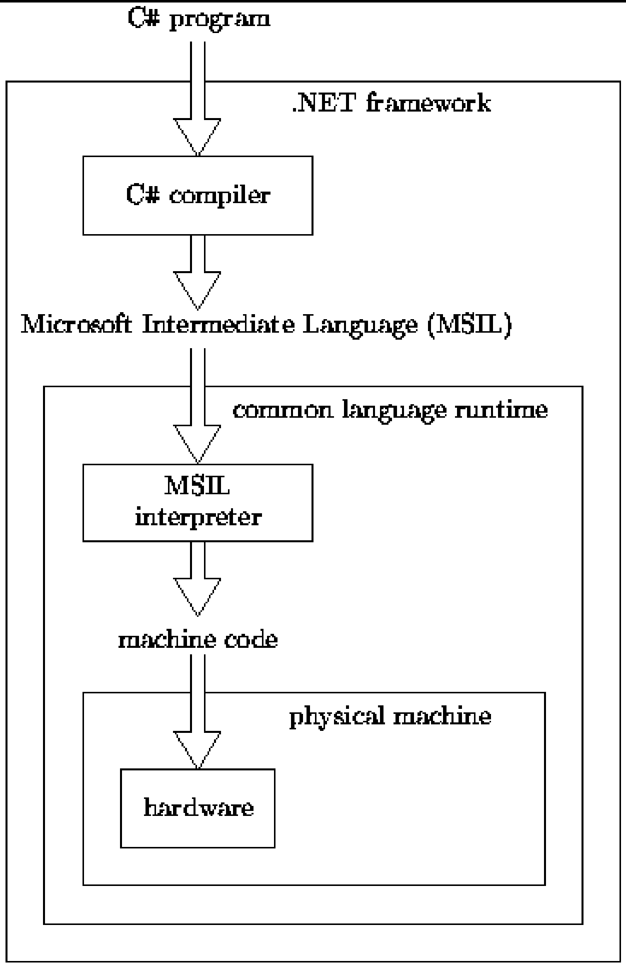

Figure 2: The *.NET* architecture diagram shows the sequence of steps involved in the execution of a C# program.

2 Features, Benefits and Advantages

Despite the fact that using Mono can enable developers to construct truly cross-platform software, it can have other advantages as well, some of which are discussed in the following sections. Mono has already been used in many production-quality commercial and open source applications[1] and is also used by many companies[2].

2.1 Why is Portability Important?

Portability is important because code written for one platform can be run on other platforms intended or unintended by the original author. The more platforms the application runs on, the larger the user-base for that application.

In general, making a fully portable cross-platform application requires careful forethought and planning. It is a decision that is best made during the beginning stages of the application's development rather than after the application has already been fully developed.

2.1.1 Business Value

Typically when developers write software applications designed to be sold, they would like to have it run on as many platforms as possible so as to sell the maximum number of licenses. This is the major under-pinning idea behind writing portable software. For example, code written in C and C++ can be ported to virtually any platform because compilers have been written for them. So for a very minimal effort, developers allow users on platforms other than his own to use his software. This is a tremendous win.

[1] http://www.mono-project.com/Software
[2] http://www.mono-project.com/Companies_Using_Mono

2.2 Technical Edge

The Mono team has developed a large amount of classes and libraries that aren't available part of the .NET framework. Some of these are [2]:

- **Mono.Directory.LDAP**: LDAP access for .NET apps.
- **Mono.Data**: Support for database access such as PostgreSQL, SQLite, Oracle, and ODBC data sources.
- **Mono.Unix**: Bindings for building POSIX applications using C#.
- **Mono.Remoting.Channels.Unix**: Unix socket-based remoting.
- **Mono.Security**: Enhanced security and crypto framework.
- **Mono.Math**: BigInteger and Prime number generation.
- **Mono.Http**: Support for creating custom, embedded HTTP servers and common HTTP handlers for your applications.
- **Mono.XML**: Extended support for XML.

In addition, Mono provides a number of profiling and code coverage tools to determine bottle-necks in code, and to see which parts of the code-base have not been fully tested yet.

2.3 Security

Mono ships with the Mono.Security namespace which provides a framework of classes for enhanced security and cryptography. Cryptography-related classes in the .NET framework can be found under a number of namespaces spread across several assemblies. In Mono, all security related classes are encapsulated under a single Mono.Security namespace.

In addition, Mono also provides extra security functionality that can be considered missing from the .NET framework via the Crimson[3] framework. The Crimson framework also contains alternative

[3]`http://www.mono-project.com/Crimson`

implementations of existing cryptographic algorithms. This gives developers a better edge because it lets them choose the best implementation based on the project's requirements and the environment in which the application is run.

2.4 Language Support

On Windows, there exists compilers for a wide variety of languages that target the virtual machine (CLR): Managed C++, Java Script, Eiffel, Component Pascal, APL, Cobol, Perl, Python, Scheme, Smalltalk, Standard ML, Haskell, Mercury and Oberon.

The beauty of .NET is that the developer has the freedom to choose the language he is most comfortable with, or is the most suitable to the task at hand.

> The CLR and the Common Type System (CTS) enable applications and libraries to be written in a collection of different languages that target the byte code. This means, for example, that if you define a class to do algebraic manipulation in C#, that class can be reused from any other language that supports the CLI. You could create a class in C#, subclass it in C++, and instantiate it in an Eiffel program. A single object system, threading system, class libraries, and garbage collection system can be shared across all these languages. [2]

Because the .NET and Mono environment are language-independent, new hires do not necessarily need to know a specific language to be able to add functionality to the application. Any language for which there exists a compiler will do. Of course, the most popular language for .NET at the time of writing is C#.

2.5 Embedded Mono

The Mono runtime can be used as a stand-alone process, or it can be embedded into applications. Embedding the Mono runtime allows applications to be extended in C# while reusing all of the existing C and C++ code.

Embedding links the Mono runtime (libmono) with the existing C/C++ application. The Mono embedded API exposes the Mono Runtime to the existing C/C++ code. Once the Mono runtime has been initialized, the code can then trigger some code, methods for example, written in, say, C#. This functionality offers many new possibilities:

- The existing C/C++ application may trigger methods that handle user interface events of a GUI application. However, core processing still remains in C/C++.
- Some of the core development may be moved to external C# libraries which is more managed than is pure C/C++ code. This provides developers with all the benefits of running code in a managed environment, like exception handling, runtime type checking, just-in-time (JIT) compilation, a rich introspection and reflection system, and type-safe libraries. [6]
- Third party libraries also written for the .NET platform may be integrated into the software written in C/C++.

3 Common Porting Issues and Solutions

Of course, not all is gold in the world of .NET and Mono. A myriad of recurring problems are encountered when even experienced developers try to write truly cross-platform applications. A few of these problems and corresponding best-practice solutions are discussed in the following sections when building software that is portable across Windows and UNIX systems using .NET and Mono.

3.1 GUI Application Development

There is a lot of concern when porting GUI-based apps to run on multiple platforms. The most common concern is whether we can preserve the native "look-and-feel" of the host operating system so that the GUI app looks and feels like it was developed natively for that operating system. To achieve this end, we would have to use GUI toolkits that are native to the target machine's operating system.

There are a number of GUI toolkit options available for developers using the Mono platform, at various degrees of completion and various degrees of functionality. [7] Some of the options available are presented here.

Mono comes packaged with Gtk#[4], a set of bindings for the popular Gtk+ GUI toolkit and assorted GNOME[5] libraries for UNIX and Windows systems. Of course, since Mono is open-source, a variety of other non-main-stream bindings have also sprung up, namely Diacanvas-Sharp and MrProject.

The Gtk# library allows developers to build fully native graphical GNOME applications using Mono. Applications built using Gtk# will run on many platforms including Linux, Windows and MacOS X. On Linux, users running the GNOME desktop environment will feel right-at-home since Gtk is the native toolkit. In short, Mono applications using Gtk# will look and function best for users who use the GNOME desktop on Linux. Mono also ships default themes for Windows to make Gtk# look and feel somewhat like a Windows application. [6] There is also a lot of effort that has been going into improving how Gtk+ based applications look on Windows. [8]

In summary, the pros and cons of using Gtk# for GUI development are:

Pros

- The Windows port of Gtk# looks and feels like a native application on Windows XP.
- Gtk# has excellent unicode support.
- Gtk# deals with internationalized environments quite well and automatically adjusts font-sizes for foreign character-sets without distorting the application's look.
- Application written using Gtk# integrate excellently with the GNOME desktop environment.
- The Gtk# API is adopted from Gtk+. Therefore its API is fairly stable and linux developers who have programmed Gtk+ previously will be familiar with the API.

Cons

[4] `http://www.mono-project.com/GtkSharp`

[5] `http://www.gnome.org/`

[6] Users who have used the GAIM instant-messaging app will testify how little variation there is with native Windows apps. GAIM uses the Gtk toolkit for its user interface.

- The documentation for Gtk# is still in its development stages and is relatively scant at the time of writing.
- Gtk# applications do not run with a native look and feel on Mac OS X. This is huge problem as Mac users are used to clean, sharp, and highly responsive GUI applications on the desktop.

If we still desire a truly native user interface on Windows, we would have to resort to using the System.Windows.Forms namespace which uses native Windows bindings. However, these bindings are not fully portable to work on Mono at the time of writing, and will therefore require the application developer to maintain two versions of the GUI front-end, one for Windows and one for Linux. This could potentially be achieved using an abstraction library to abstract out platform-specific code.

This idea can be extended to use each platform's native UI for every platform the application decides to support. This can be achieved by completely decoupling core application code from UI code. That way, we could use Gtk# on Linux, WinForms (the System.Windows.Forms namespace) on Windows, and Cocoa# (the native Mac OS X GUI toolkit) on the Mac. This would increase the amount of code that needs to be managed, but would ensure a very native look and feel for each individual platform.

3.2 Internationalization

Internationalization is a major issue as application developers try to write programs that can be localized for a specific set of users. Many applications, code, documentation, tooltips, and error messages written today are written for English speaking users only and don't adapt to other locales, languages, and customs.

The traditional way, on Linux, of using multiple language strings is the `gettext` library. Mono introduces a Mono.Unix namespace which is the recommended way of working with `gettext` to translate an application's strings. The new namespace provides the `Catalog` class which is a wrapper to the `libintl` library thereby providing key-based message translation capabilities.

Internationalization is something that needs to be thought about from the very beginning rather than during the later stages of the application's development.

3.3 File Paths

One of the most common problems that people face when porting applications from Windows to Linux using Mono are paths. Most Windows developers are used to a case-insensitive file system and as such refer to file names like "geometrytools.cpp" as "GeometryTools.cpp" or "geometrytools.CPP" elsewhere in the code. This, of course, does not work on UNIX file-systems as most UNIX file-systems are case-sensitive, causing the program to throw a FileNotFound exception . As such, two files with the same name but different cases are considered altogether different files on most UNIX systems.

Windows developers also sometimes hard-code path, directory, and drive separators within their code. This, of course, does not work on UNIX as ";", "\" and ":" are valid parts of a filename within a directory. For example, applications that hard-code the path "logs\access.log" in their code will not work on UNIX which will, instead of referencing the access.log file within the logs directory, will references the file called "logs\access.log" within the current directory. Similarly, "C:\myfile.txt" is a valid filename on UNIX-like systems.

Developers can easily solve this problem by using Path.DirectorySeparator and Path.Combine to obtain the directory and path separators respectively for the current system. The traditional way to find these kind of bugs in the application would be to run the test-suite and the application, and then to listen for FileNotFound exceptions. Another symptom is the creation of empty files because the data was redirected to a different folder than the one intended. Surely, this method of finding path errors is a very time consuming process and is not guaranteed to work all the time. Further, this strategy assumes that the code is available to developers to fix if a bug is encountered. This is not the case when using third-party libraries.

To resolve this issue, Mono introduces a new portability layer without requiring changes to the code itself. The new portability framework is enabled by setting the environment variable `MONO_IOMAP` to one of the following values:

- **case**: makes all file system access case insensitive.
- **drive**: strips drive name from pathnames.
- **all**: enables both case and drive.

The directory separator mapping is also turned on automatically when any of the above options are enabled.

Of course, enabling the portability layer introduces an additional overhead as Mono performs extra work to cope with the file-system in use. The most prudent strategy for best performance is to already start with truly portable code and to turn off the Mono portability layer altogether.

4 Porting Strategy

Developers can continue to use Microsoft Visual Studio, develop their applications on Windows, and then manually copy over the binaries produced by Visual Studio over to UNIX. These binaries are compatible with and can be executed by Mono directly. Alternatively, developers could set up a network share that can be commonly accessed by both Windows and UNIX. Developers already familiar with UNIX's development environment and editors can use the packaged command line tools to develop portable applications.

Developers must ensure the use of System.IO.Path.DirectorySeparatorChar character when concatenation of paths is needed. (On Windows, the directory path separator is "\" while on Linux it is "/".) Even better is to use the System.IO.Path.Combine method to combine pathnames. All file paths must be treated in a case sensitive manner. That is, the files "readme" and "README" are two different files. As much as possible, the reliance on the aforementioned IO Remapping functionality should be avoided as remapping introduces a slight performance penalty. Finally, environment variables like PATH must have their directories separated by System.IO.Path.PathSeparator to ensure portability between various operating systems. (On Windows, the PATH has directories separated by a semicolon ";" and on Linux by a colon ":".)

Absolute paths are treated differently as well between Windows and Linux. On Linux, any path that begins with a forward slash "/" is an absolute path, while Windows requires a drive letter to qualify a path as absolute. In general, absolute path names must be avoided in cross-platform compliant applications.

Further, to keep code running in as many platforms as possible, developers should keep all code Endian-independent by not assuming anywhere the order of bytes.

A few porting tools are available as part of the Mono installation package to ease the porting process. One such tools is `prj2make` which converts Visual Studio project and solution files to UNIX Makefiles. This serves as a quick first-pass conversion when porting from Windows to UNIX. However, `prj2make` is not production-ready yet and does not work for all project files. `prj2make` is also unable to handle case differences in filenames. So if a project file references a file called "Handler.cs" when the file on disk is actually "handlers.cs", the conversion will fail. It is up to the developer to fix these case issues manually when the compilation fails.

5 Concluding Summary

In conclusion, technologies such as .NET and Mono can be quite useful in shortening the porting process by providing tools and utilities that aid the developer in the porting process. .NET introduces a framework wherein software can be compiled on any operating system. The resulting semi-compiled byte-code can be executed on any platform for which a port of Mono is available. Even at the time of writing, all major operating systems and machine architectures (including 64-bit) have been covered.

When using .NET, the developer will need to ensure not to include any operating-system specific constructs in his code. Instead, he should use the various classes and constants provided to maintain platform agnosticism. The developer must also assume that the file-system is case-sensitive even if he is currently working on a case-insensitive system to ensure future portability. The developer should restrict himself to using the Mono.Unix namespace too internationalize language strings. Lastly, developers should keep all code Endian-independent to ensure portability across architectures that vary in the order of bytes.

In conclusion, as long as developers have a clear porting strategy and constantly keep in mind the recurring issues, the .NET/Mono combination is an excellent solution for developing truly cross-platform software.

References

[1] Mono Main Page. *What is Mono?*. Last retrieved April 16, 2007. `http://www.mono-project.com/Main_Page`.

[2] Mono General FAQ. Last retrieved April 16, 2007. `http://www.mono-project.com/FAQ:_General`.

[3] Mono Supported Platforms. *Supported Architectures*. Last retrieved April 16, 2007. `http://www.mono-project.com/Supported_Platforms`.

[4] .NET Framework Architecture. *Simplified Mono Architecture*. Last retrieved May 6, 2007. `http://www.mono-project.com/.NET_Framework_Architecture`.

[5] Easton, M.J and King, Jason. *Cross-Platform .NET Development: Using Mono, Portable.NET, and Microsoft .NET*. Apress Publishing. September 2004.

[6] Embedding Mono. *How Embedding Works* Last retrieved April 17, 2007. `http://www.mono-project.com/Embedding_Mono`.

[7] GUI Toolkits Overview for Mono. Last retrieved April 18, 2007. `http://primates.ximian.com/~miguel/toolkits.html`.

[8] Improving Gtk on Win32. Last retrieved April 23, 2007. `http://www.mono-project.com/ImprovingGtkWin32`.